A Timed Semantics of Orc

Ian Wehrman, David Kitchin, William R. Cook, Jayadev Misra

The University of Texas at Austin

Abstract

Orc is a kernel language for structured concurrent programming. Orc provides three powerful combinators that define the structure of a concurrent computation. These combinators support sequential and concurrent execution, and concurrent execution with blocking and termination.

Orc is particularly well-suited for *task orchestration*, a form of concurrent programming with applications in workflow, business process management, and web service orchestration. Orc provides constructs to orchestrate the concurrent invocation of services while managing time-outs, priorities, and failures of services or communication.

Our previous work on the semantics of Orc has focused on its asynchronous behavior. The inclusion of time or the effect of delay on a computation had not been modeled. In this paper, we define an operational semantics of Orc that allows reasoning about delays, which are introduced explicitly by time-based constructs or implicitly by network delays. We develop a number of identities among Orc expressions and define an equality relation that is a congruence. We also present a denotational semantics in which the meaning of an Orc program is a set of traces, and show that the two semantics are equivalent.

1. Introduction

Orc is a language for structured concurrent programming. It is based on the premise that structured concurrent programs should be developed much like structured sequential programs, by decomposing a problem and combining the solutions with combinators of the language. Naturally, Orc combinators support concurrency: parallel subcomputations, spawning of computations and blocking or termination of subcomputations.

Expressions in an Orc program are either primitive or a combination of two expressions. A primitive expression is a call to an existing service, a *site*, to perform its computations and return a result to the caller. There are only three combinators for Orc expressions, which allow sequential and concurrent executions of expressions, and concurrent execution with termination.

Orc is particularly well-suited for task orchestration, a form of concurrent programming in which multiple services are invoked to achieve a goal while managing time-outs, priorities, and failures of services or communication. Unlike traditional concurrency models, orchestration introduces an *asymmetric* relationship between a program and the services that constitute its environment. An orchestration invokes and receives responses from the external services, which do not initiate communication. In this paper, we illustrate the use of Orc in implementing some traditional concurrent computation patterns; larger examples have also been developed [20,9]. Orc has also been used to study service-level agreements for composite web services [24].

[1] Work of the second and third authors is partially supported by National Science Foundation grant CCF-0448128.

Time is an essential aspect of many orchestrations—time-critical business workflows, for example, are naturally expressed as orchestrations [1]. Time is introduced in Orc implicitly by delays resulting from remote service calls, and explicitly by the site *Rtimer*, which waits a given amount of time when invoked before continuing execution. Previous accounts of the semantics of Orc [14,20,23,13] have not covered the semantics of time.

In Section 4, an operational semantics of Orc is given that includes time. The semantics shown here is based on an asynchronous semantics of Orc [14]. The transition relation of the asynchronous operational semantics is extended to include the time at which an event occurs. The corresponding executions are changed from a sequence of events to a sequence of time-event pairs. The semantics allows multiple events to occur at a single instant of time. An important feature of the semantics presented here is that time can be considered either discrete or continuous.

We have shown for the asynchronous semantics that equality of trace sets defines a congruence on programs, in that programs with equivalent trace sets are interchangeable [14]. We establish the same result for timed semantics. Additionally, we give a number of identities in Section 7, similar to those of Kleene algebra [16], that hold for the Orc combinators in the timed semantics. In Section 8, we show that traces form a denotation, which allows us reason about the operational behavior of an Orc expression compositionally. In particular, the denotational semantics shows that the traces of a recursively defined expression can be computed as the limit of a sequence of traces. In Section 9, we show that the operational and denotational semantics agree.

Detailed proofs of all the results stated in this paper can be found in a companion technical report, Wehrman, et. al [27]. Portions of Sections 2 and 3 appeared previously in [14].

2. Overview of Orc

An Orc program consists of a *goal* expression and a set of definitions. The goal expression is evaluated in order to run the program. The definitions are used in the goal and in other definitions.

An expression is either primitive or a combination of two expressions. A primitive expression is a call to an existing service, a *site*, to perform its computations and return a result; we describe sites in Section 2.1. Additionally, **0** is a primitive described in Section 4.1, which has no observable transitions. Two expressions can be combined to form a composite expression using Orc combinators; we describe the combinators in Section 2.2. We allow expressions to be named in a definition, and these names may then be used in other expressions. Naming permits us to define an expression recursively by using its own name in the definition. Definitions and recursion are treated in Section 2.3. We give a complete formal syntax in Figure 2 of Section 2.4.

During its evaluation, an Orc expression calls sites and publishes values. Below, we describe the details of calls and publications.

2.1. *Sites*

A primitive Orc expression is a *site call* $M(\bar{p})$, where M is a site name and $\bar{p}$ a list of actual parameters. A site is an external program, 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. For example, evaluation of $CNN(d)$, where CNN is a news service site and d is a date, calls CNN with parameter value d; if CNN responds (with the news page for the specified date), the response is published.

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

Figure 1 lists a few sites that are fundamental to effective programming in Orc (in the figure, a *signal* is a unit value and has no additional information). *Signal* is a site which responds immediately with a signal (it is the same as $if(true)$). Site *Rtimer* is used to introduce delays and impose time-outs, and is essential for time-based computations. Examples appear in Section 3.

$let(x, y, \cdots)$ Returns argument values as a tuple.

$if(b)$ Returns a signal if b is *true*, and otherwise does not respond.

$Rtimer(t)$ Returns a signal after exactly t, $t \geq 0$, time units.

Fig. 1. Fundamental Sites

2.2. *Combinators*

There are three combinators in Orc for combining expressions f and g: symmetric parallel composition, written as $f \mid g$; sequential composition with respect to variable x, written as f $>x>$ g; and asymmetric parallel composition with respect to variable x, written as f $<x<$ g.

To evaluate $f \mid g$, we evaluate f and g independently. The sites called by f and g are the ones called by $f \mid g$ and any value published by either f or g is published by $f \mid g$. There is no direct communication or interaction between these two computations. For example, evaluation of $CNN(d) \mid BBC(d)$ initiates two independent computations; up to two values will be published depending on which sites respond.

In f $>x>$ g, expression f is evaluated and each value published by it initiates a fresh instance of g as a separate computation. The value published by f is bound to x in g's computation. Evaluation of f continues while (possibly several) instances of g are run. If f publishes no value, g is never instantiated. The values published by f $>x>$ g are the ones published by all the instances of g (values published by f are consumed within f $>x>$ g). This is the only mechanism in Orc similar to spawning threads.

As an example, the following expression calls sites *CNN* and *BBC* in parallel to get the news for date d. Responses from either of these calls are bound to x and then site *email* is called to send the information to address a. Thus, *email* may be called 0, 1 or 2 times.

$$(CNN(d) \mid BBC(d)) \; {>}x{>} \; email(a, x)$$

Expression $f \gg g$ is short-hand for f $>x>$ g, where x is not free in g.

As a short example of time-based computation, $Rtimer(2) \gg M$ delays calling site M for two time units, and $M \mid (Rtimer(1) \gg M) \mid (Rtimer(2) \gg M)$ makes three calls to M at unit time intervals.

To evaluate (f $<x<$ g), start by evaluating both f and g in parallel. Evaluation of parts of f which do not depend on x can proceed, but site calls in which x is a parameter are suspended until x has a value. If g publishes a value, then x is assigned the (first such) value, g's evaluation is terminated and the suspended parts of f can proceed. The values published by (f $<x<$ g) are the ones published by f. Any response received for g after its termination is ignored. This is the only mechanism in Orc to block or terminate parts of a computation.

As an example, in $((M \mid N(x))$ $<x<$ $R)$ sites M and R are called immediately (thus, M is called immediately, even before x may have a value). Once R responds, x is assigned a value and $N(x)$ is then called. Contrast the following expressions; in the first one *email* is called at most once, whereas the second one (shown earlier) may call *email* twice.

$$email(a, x) \; {<}x{<} \; (CNN(d) \mid BBC(d))$$
$$(CNN(d) \mid BBC(d)) \; {>}x{>} \; email(a, x)$$

2.3. *Definitions and Recursion*

Declaration $E(\bar{x}) \;\underline{\Delta}\; f$ defines expression E whose formal parameter list is $\bar{x}$ and body is expression f. We assume that only the variables $\bar{x}$ are free in f. A call $E(\bar{p})$ is evaluated by replacing the formal parameters $\bar{x}$ by the actual parameters $\bar{p}$ in the body of the definition f. Sites are called by value, while definitions are called by name.

A definition may be recursive (or mutually recursive): a call to E may occur in f, the body of the expression, yielding a recursively defined expression. Such expressions are used for encoding bounded as well as unbounded computations. Below, *Metronome* publishes a signal every time unit starting immediately.

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

2.4. *Formal Syntax*

$$\begin{array}{rcll} f,g,h & \in & \mathit{Expression} & ::= M(\bar{p}) \;\boldsymbol{|}\; E(\bar{p}) \;\boldsymbol{|}\; f >x> g \;\boldsymbol{|}\; f \mid g \;\boldsymbol{|}\; f <x< g \\ p & \in & \mathit{Actual} & ::= x \;\boldsymbol{|}\; m \\ & & \mathit{Definition} & ::= E(\bar{x}) \;\; \underline{\Delta} \;\; f \end{array}$$

Fig. 2. Syntax of Orc

The formal syntax of Orc is given in Figure 2. (Previous presentations of Orc have used the notation f **where** $x :\in g$ instead of $f <x< g$.) Here M is the name of a site and E a defined expression. An actual parameter p may be a variable x or a value m, and $\bar{p}$ denotes a list of actual parameters. If the parameter list is empty in $M(\bar{p})$ or $E(\bar{p})$, we simply write M or E.

Notation The combinators are listed below in decreasing order of precedence, so $f <x< g \mid h$ means $f <x< (g \mid h)$, and $f >x> g \mid h$ means $(f >x> g) \mid h$.

3. Examples

Time-out

The following expression publishes the first value published by f if it is available before time t, otherwise publishes 3. It evaluates f and $\mathit{Rtimer}(t) \gg \mathit{let}(3)$ in parallel and takes the first value published by either:

$$\mathit{let}(z) <z< (f \mid \mathit{Rtimer}(t) \gg \mathit{let}(3))$$

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, the value of x is irrelevant. Below, z is the pair (x,b).

$$\mathit{let}(z) \;\; <z< \;\; (\; M >x> \mathit{let}(x, \mathit{true}) \;\;|\;\; \mathit{Rtimer}(t) >x> \mathit{let}(x, \mathit{false}) \;)$$

Fork-join Parallelism

In concurrent programming, one often needs 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. Below, we define *forkjoin* to call sites M and N in parallel and publish their values as a tuple after they both complete their executions.

$$\mathit{forkjoin} \;\; \underline{\Delta} \;\; (\mathit{let}(x,y) <x< M) <y< N$$

The following expression publishes N's response as soon as possible, but after at least one time unit. This is similar to a fork-join on $\mathit{Rtimer}(1)$ and N.

$$\mathit{Delay} \;\; \underline{\Delta} \;\; (\mathit{Rtimer}(1) \gg \mathit{let}(y)) <y< N$$

Synchronization

There is no special machinery for synchronization in Orc; a **where** expression provides the necessary ingredients for programming synchronizations. Consider $M \gg f$ 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. We evaluate *forkjoin*, and start $f \mid g$ after *forkjoin* publishes.

$$\mathit{forkjoin} \gg (f \mid g)$$

Priority

Call sites M and N simultaneously. If M responds within one time unit, take its response, otherwise pick the first response. Using *Delay* defined earlier,

$$\mathit{let}(x) <x< (M \mid \mathit{Delay})$$

Nondeterministic Choice

Process algebras often include a *nondeterministic choice* operator $\oplus$, where expression $P \oplus Q$ may behave as either process P or process Q. To encode this construct in Orc, we observe that in asymmetric composition the choice of a first value is nondeterministic if several values are published simultaneously.

$$\begin{array}{l} \mathit{if}(\mathit{flag}) \gg P \mid \mathit{if}(\neg \mathit{flag}) \gg Q \\ \quad {<}\mathit{flag}{<}\ (\mathit{let}(\mathit{true}) \mid \mathit{let}(\mathit{false})) \end{array}$$

Iterative Process and Process Networks

A process in a typical network-based computation repeatedly reads a value from a channel, computes with it and writes the result to another channel. Below, c and e are channels, and $c.\mathit{get}$ and $e.\mathit{put}$ are the methods to read from c and write to e. We treat these methods as sites. Below, $P(c, e)$ repeatedly reads from c and writes to e, and $\mathit{Net}(c, d, e)$ is a network of two such processes which share the output channel.

$$\begin{array}{rclll} P(c,e) & \underline{\Delta} & c.\mathit{get} & {>}x{>} & \mathit{Compute}(x) \\ & & & {>}y{>} & e.\mathit{put}(y) \\ & & & \gg & P(c,e) \\ \mathit{Net}(c,d,e) & \underline{\Delta} & \multicolumn{3}{l}{P(c,e) \mid P(d,e)} \end{array}$$

Parallel-or

A classic problem in non-strict evaluation is *parallel-or*. Suppose sites M and N publish booleans. We desire an expression that publishes *true* as soon as either site returns *true*, and *false* only if both return *false*. Otherwise, the expression never publishes. In the following solution, site $\mathit{or}(x, y)$ returns $x \vee y$. Define $\mathit{ift}(b)$ to return *true* if b is true, and to not respond otherwise: $\mathit{ift}(b)\ \underline{\Delta}\ \mathit{if}(b) \gg \mathit{let}(\mathit{true})$.

$$\begin{array}{rl} (\mathit{let}(z)\ {<}z{<}\ \mathit{ift}(x) \mid \mathit{ift}(y) \mid \mathit{or}(x,y)) & {<}x{<}\ M \\ & {<}y{<}\ N \end{array}$$

4. Timed Operational Semantics

The operational semantics of Timed Orc is a labeled transition system, which is based on the operational semantics of Orc without time [14,22]. As is common in small-step operational semantics, the language must be extended to represent intermediate states. We extend the syntax of Orc to include the expression $?k$ to denote an instance of a site call that has not yet returned a value, where k identifies the call instance. The labels of the transition system are time-event pairs (t, a). The transition relation $f \overset{t,a}{\rightarrow} f'$, defined in Figure 3, states that expression f may transition with event a to expression f', where the transition occurs exactly t time units after its evaluation starts.

Events are either *publication* events, written $!m$, or *internal* events, written τ. Publication events correspond to the communication of value m to the environment during a transition. Internal events correspond to state changes not intended to be observable by the environment. We refer to both publication and internal events as *base* events.

The times in the transition relation are *relative* to the start of evaluation of the expression. Furthermore, $f \overset{t,a}{\rightarrow} f'$ specifies that no other events have occurred in the t units that have passed since the beginning of the evaluation of f. Times may be drawn from any totally-ordered set with a least element, such as the non-negative reals or the natural numbers. In this document we take times to be non-negative reals.

Notation Henceforth, expressions are denoted by f, g, h; variables by x, y, z; events by a, b; and times by t, s. Sets of objects are denoted by the upper-case versions of their corresponding letters. Parameters, which are either variables or values, are denoted by p. Substitution application is denoted by $[m/y].f$, defined formally in Figure 4.

$$\frac{[E(x) \;\underline{\Delta}\; f] \in \mathcal{D}}{E(p) \overset{0,\tau}{\to} [p/x].f} \quad (\textsc{Def}) \qquad \frac{f \overset{t,a}{\to} f' \qquad a \neq !m}{f >x> g \overset{t,a}{\to} f' >x> g} \quad (\textsc{Seq1N})$$

$$\frac{k \in \Sigma(M,m)}{M(m) \overset{0,\tau}{\to} ?k} \quad (\textsc{Call}) \qquad \frac{f \overset{t,!m}{\to} f'}{f >x> g \overset{t,\tau}{\to} (f' >x> g) \mid [m/x].g} \quad (\textsc{Seq1V})$$

$$\frac{(t,m) \in k}{?k \overset{t,!m}{\to} \mathbf{0}} \quad (\textsc{Return}) \qquad \frac{f \overset{t,a}{\to} f'}{f <x< g \overset{t,a}{\to} f' <x< g^t} \quad (\textsc{Asym1})$$

$$\frac{f \overset{t,a}{\to} f'}{f \mid g \overset{t,a}{\to} f' \mid g^t} \quad (\textsc{Sym1}) \qquad \frac{g \overset{t,!m}{\to} g'}{f <x< g \overset{t,\tau}{\to} [m/x].f^t} \quad (\textsc{Asym2V})$$

$$\frac{g \overset{t,a}{\to} g'}{f \mid g \overset{t,a}{\to} f^t \mid g'} \quad (\textsc{Sym2}) \qquad \frac{g \overset{t,a}{\to} g' \qquad a \neq !m}{f <x< g \overset{t,a}{\to} f^t <x< g'} \quad (\textsc{Asym2N})$$

Fig. 3. Timed Semantics of Orc

$$\begin{aligned}
[m/y].(?k) &= ?k \\
[m/y].(M(p)) &= \begin{cases} M(m) & \text{if } p = y \\ M(p) & \text{otherwise} \end{cases} \\
[m/y].(E(p)) &= \begin{cases} E(m) & \text{if } p = y \\ E(p) & \text{otherwise} \end{cases} \\
[m/y].(f \mid g) &= ([m/y].f) \mid ([m/y].g) \\
[m/y].(f >x> g) &= \begin{cases} ([m/y].f) >x> g & \text{if } x = y \\ ([m/y].f) >x> ([m/y].g) & \text{otherwise} \end{cases} \\
[m/y].(f <x< g) &= \begin{cases} f <x< ([m/y].g) & \text{if } x = y \\ ([m/y].f) <x< ([m/y].g) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 4. Definition of Substitution Application

4.1. *Site Calls and Responses*

Sites are the fundamental units of computation in Orc, and can be thought of as either unreliable remote services (e.g., *BBC*), or as locally defined procedures with predictable behavior (e.g., *if*). We refer to the former sites as *remote* and the latter as *local.*

The (Call) rule in Figure 3 describes the operational semantics of site calls. [2] It specifies that expression $M(m)$—the invocation of site M with value m—performs an internal event at relative time 0 (i.e., without delay) and transitions to an intermediate expression $?k$. We write $\Sigma(M,m)$ for the set of *handles* that correspond to expression $M(m)$. Each handle describes a possible behavior of site M when it is called with value m. We also call $?k$, the expression corresponding to handle k.

Informally, a handle specifies the relative times at which particular values could potentially be returned by a site call, and also the possibility of perpetual non-response. A handle is a set of pairs (t,m), where t is

[2] We restrict discussion to the semantics of sites and definitions with a single argument. Multiple arguments are easily handled by adding tuples to the language.

a time and m is a value, denoting that m may be returned at time t as a response. Additionally, a handle may also include a distinguished element $\omega \notin \mathcal{T}$, which indicates non-response. Hence, for the set of relative times $\mathcal{T}$ and universe of values $\mathcal{V}$, handle k satisfies

$$k \subseteq (\mathcal{T} \times \mathcal{V}) \cup \{\omega\}.$$

The (Return) rule describes the behavior of handles as a set of potential responses in time. If $(t, m) \in k$, then $?k$ may transition after t units with event $!m$ to $\mathbf{0}$, an expression which has no observable transitions. If $\omega \in k$, then it is possible that the handle will never respond, in which case the call blocks indefinitely. If a handle specifies more than one potential action (i.e., response or non-response), any one of the values may be returned at the associated time.

Local Sites

Local sites have predefined and predictable behavior. Consequently, we can define $\Sigma(M, m)$ completely for a local site M and any value m. (In the following definition, we write $\cdot$ for signal, a unit value.) Recall that $\Sigma(M, m)$ is a set of handles, where each handle is a set of pairs (t, v) or ω. For the sites in Figure 5, there is exactly one handle for each site for a specific parameter value.

$$\begin{aligned} \Sigma(\mathit{let}, m) &= \{\{(0, m)\}\} & \Sigma(\mathit{if}, \mathit{false}) &= \{\{\omega\}\} \\ \Sigma(\mathit{Signal}) &= \{\{(0, \cdot)\}\} & \Sigma(\mathit{if}, \mathit{true}) &= \{\{(0, \cdot)\}\} \\ \Sigma(\mathit{Rtimer}, t) &= \{\{(t, \cdot)\}\} \end{aligned}$$

Fig. 5. Environment Requirements for Local Sites

The definitions imply that $\mathit{let}(m)$ engages in $!m$ immediately and that $\mathit{Rtimer}(t)$ signals after exactly t time units. Additionally, the primitive expression $\mathbf{0}$ can now be defined as the handle $?k$, where $k = \{\omega\}$.

4.2. *Time-shifted Expressions*

A time-shifted expression, written f^t, is the expression that results from f after t units have elapsed *without occurrence of an event.* When it is not possible for t time units to elapse without f engaging in an event we write $f^t = \bot$, where $\bot$ is an unreachable expression described later. The time-shifted expression f^t, for $t \geq 0$, is defined in Figure 6 based on the structure of f.

$$\begin{aligned} ?k^t &= ?(\{(s, m) \mid (t + s, m) \in k\} \cup (k \cap \{\omega\})) \\ M(x)^t &= M(x) \\ M(m)^t &= \begin{cases} M(m) & \text{if } t = 0 \\ \bot & \text{otherwise.} \end{cases} \\ E(p)^t &= \begin{cases} E(p) & \text{if } t = 0 \\ \bot & \text{otherwise.} \end{cases} \\ (f \mid g)^t &= f^t \mid g^t \\ (f > x > g)^t &= f^t > x > g \\ (f < x < g)^t &= f^t < x < g^t \end{aligned}$$

Fig. 6. Definition of Time-shifted Expressions

The first three cases, for each of the combinators, are easy to justify informally. Expression $M(x)^t$, where x is a variable, is simply $M(x)$ because the site cannot be invoked until the parameter has a value. Expression $M(m)$, where m is a value, must be invoked at time 0; therefore, $M(m)^0 = M(m)$, whereas $M(m)^t = \bot$ for $t > 0$. The time-shifted handle $?k^t$ may publish m at time s iff $?k$ may publish m at $t + s$; and $?k^t$ includes

ω iff $?k$ does. We take $?\emptyset$ to mean $\bot$. Like site calls, defined expressions must be evaluated immediately because $E(p)^t = \bot$ for $t > 0$.

The definitions for $M(x)^t$ and $M(m)^t$ in Figure 6 also encompass local sites $\mathit{if}(\mathit{true})^t$, Signal^t, $\mathit{let}(m)^t$, etc. Of particular importance is *Rtimer*. Consider the handle $?k$ that results from a call to $\mathit{Rtimer}(3)$. It is easily seen that $?k^2 = ?j$, where $?j$ is a handle resulting from a call to $\mathit{Rtimer}(1)$, i.e., $\mathit{Rtimer}(3)$ behaves like $\mathit{Rtimer}(1)$ after 2 times units have elapsed.

We note the following facts about time-shifted expressions, which can be proved by structural induction on f.

$f^0 = f$
$(f^s)^t = f^{s+t}$
$f^s \overset{t,a}{\to} h \;\equiv\; f \overset{s+t,a}{\to} h$

Reachable Expressions

In some cases, it is not possible for t units of time to elapse without occurrence of an event. For example, it is not possible for 1 unit to elapse without an event after the start of evaluation of $\mathit{let}(1)$ because the site call must occur without any delay. Similarly, if $?k$ results from a call to $\mathit{Rtimer}(2)$, it is not possible for 3 units to elapse without event, i.e., $?k^3 = \bot$.

Any expression which has $\bot$ as a constituent is defined to be $\bot$. Such an expression is *unreachable*, whereas typical Orc expressions are *reachable*. In particular, there is no event (t, a) for which $\bot \overset{t,a}{\to}$. The transition $f \overset{t,a}{\to} \bot$ for a reachable expression f denotes that f does not engage in the given transition.

4.3. *Combinator Rules*

We now describe the rules in Figure 3 that pertain to the three combinators. From $f \overset{t,a}{\to} f'$, we can infer with rule (Sym1) that $f \mid g \overset{t,a}{\to} f' \mid g^t$. Here, g is time-shifted to g^t because t time units have elapsed without an event by g. Note that g^t could be $\bot$; in that case, the rule cannot be applied because the corresponding transition is not counted as part of an execution (see Section 5). Similar remarks apply to (Sym2), (Asym1), (Asym2V) and (Asym2N).

When f publishes a value $f \overset{t,!m}{\to} f'$, rule (Seq1V) creates a new instance of the right side, $[m/x].g$, the expression in which all free occurrences of x in g are replaced by m.[3] The publication $!m$ is hidden, and the entire expression performs a τ event. Note that f and all instances of g are executed in parallel. Because multiple events may occur at the same time instant, it is not guaranteed that the values published by the first instance will precede the values of later instances.

Asymmetric parallel composition is similar to parallel composition, except when g publishes a value m. In this case, rule (Asym2V) terminates g and x is bound to m in f. One subtlety of these rules is that f may contain both active and blocked subprocesses – any site call that uses x is blocked until g publishes.

Expressions are evaluated using call-by-name in the (Def) rule. We assume a single global set of definitions $\mathcal{D}$.

Example We show below a sequence of one-step evaluations of the expression $(\mathit{Rtimer}(3) \gg M(x)) \mid N$. The resulting expression is in normal form.

$(\mathit{Rtimer}(3) \gg M(x)) \mid N$
$\overset{0,\tau}{\to}$ $\{$(Sym2), (Call), $k \in \Sigma(N)\}$
$(\mathit{Rtimer}(3) \gg M(x)) \mid ?k$
$\overset{0,\tau}{\to}$ $\{$(Sym1), (Seq1N), (Call), $j = \{(3, \cdot)\}\}$

[3] Recall that $f \gg g$ is short for $f > x > g$ for some variable x not free in g. So if $f \overset{t,!m}{\to} f'$ then, by rule (Seq1V), $f \gg g \overset{t,\tau}{\to} (f' \gg g) \mid g$.

$$
\begin{array}{ll}
 & (?j \gg M(x)) \mid ?k \\
\overset{2,!n}{\rightarrow} & \{(\textsc{Sym2}),\ (\textsc{Return}),\ \text{assuming } (2,n) \in k\} \\
 & (?j^2 \gg M(x)) \mid \mathbf{0} \\
\overset{1,\tau}{\rightarrow} & \{(\textsc{Sym1}),\ (\textsc{Seq1V}),\ (\textsc{Return}),\ ?j^2 = \{(1,\cdot)\}\} \\
 & ((\mathbf{0} \gg M(x)) \mid M(x)) \mid \mathbf{0}
\end{array}
$$

5. Executions and Traces

In this section, we formalize the notions of *executions* and *traces* for expressions. An execution of f is a sequence of timed events in which f may engage. A trace is an execution with the τ events removed.

The execution relation $\Rightarrow$ is derived from the reflexive and transitive closure of the transition relation $\rightarrow$ of Figure 3. However, we need to shift the times in forming the transitive closure. Given $f \overset{(s,a)}{\rightarrow} f'$ and $f' \overset{(t,b)}{\rightarrow} f''$, we can not claim that $f \overset{(s,a)(t,b)}{\Rightarrow} f''$, because b occurs $s+t$ units after the evaluation of f starts. We define u_t as the sequence that results from increasing each time component of u by t. The definition of u_t is also lifted to sets pointwise: $U_t = \{u_t \mid u \in U\}$.

Define relation $\Rightarrow$ as the reflexive-transitive closure of relation $\rightarrow$ except that the time components accumulate.

$$
f \overset{\epsilon}{\Rightarrow} f \quad (\textsc{Ex-Refl}) \qquad\qquad \frac{f \overset{(t,a)}{\rightarrow} f'',\ f'' \overset{u}{\Rightarrow} f'}{f \overset{(t,a)u_t}{\Rightarrow} f'} \quad (\textsc{Ex-Trans})
$$

Call u an *execution* of f if $f \overset{u}{\Rightarrow} f'$ for some $f' \neq \bot$. Note that the empty sequence ϵ is an execution of any expression by rule (Ex-Refl).

The definition of executions requires $f' \neq \bot$ so that all intermediate expressions in an execution (such as f'') are reachable—if any intermediate expression is unreachable, the final expression, f', would be unreachable because $\bot$ has no transitions.

Example The example of Section 4.3, $(\mathit{Rtimer}(3) \gg M) \mid N$, has an execution shown below:

$u = (0,\tau)\,(0,\tau)\,(2,\,!n)\,(3,\tau)$

A *trace* $\overline{u}$ is obtained from execution u by removing each internal event (t,τ). The definition is also lifted pointwise to sets: $\overline{U} = \{\overline{u} \mid u \in U\}$.

Example Execution u and its trace $\overline{u}$ are shown below:

$u = (0,\tau)\,(0,\tau)\,(2,\,!a)\,(3,\tau)$

$\overline{u} = \qquad\qquad (2,\,!a)$

Notation The *execution set* and *trace set* of f are written $[\![\, f \,]\!]$ and $\langle\!\langle f \rangle\!\rangle$ respectively:

$$[\![\, f \,]\!] = \{u \mid f \overset{u}{\Rightarrow} f', \text{for some } f'\}, \text{ and } \langle\!\langle f \rangle\!\rangle = \overline{[\![\, f \,]\!]}.$$

We define $f \sim g$ to mean $[\![\, f \,]\!] = [\![\, g \,]\!]$ and $f \cong g$ to mean $\langle\!\langle f \rangle\!\rangle = \langle\!\langle g \rangle\!\rangle$. We will show that $\sim$ and $\cong$ are congruence relations, so that related expressions can be replaced by each other in all contexts. This claim, however, is not true with the theory developed so far. For example, we can prove that $\mathbf{0} \sim \mathit{let}(x)$ because neither has an observable transition. Yet these two expressions display different behaviors in the same context: $\mathit{let}(1) >x> \mathbf{0}$ never publishes, whereas $\mathit{let}(1) >x> \mathit{let}(x)$ always publishes. Our goal is for traces to represent the observable behavior of an expression, thus the semantics must be extended to distinguish these two cases.

6. Substitution Events

We introduce another kind of event, called a *substitution event*, to represent the binding of a value to a free variable in an expression. Substitution events have the form $(t, [m/x])$, where m is a value and x is a variable. The following transition rule introduces substitution events at the top level of expression derivations.

$$f \overset{t,[m/x]}{\rightarrow} [m/x].(f^t) \qquad (\textsc{Subst})$$

Henceforth, we write $[m/x].f^t$ to mean $[m/x].(f^t)$, i.e., the time-shift operator binds more strongly than substitution. Thus, using rule (Subst) and the definitions of time shifting and substitution we get

$$f \mid g \overset{t,[m/x]}{\rightarrow} [m/x].f^t \mid [m/x].g^t.$$

A substitution event differs from the base events described in Section 4 in a crucial way: the rules in Figure 3 are defined only over base events. Therefore, given that $f \overset{t,[m/x]}{\rightarrow} [m/x].f^t$, (Sym1) can *not* be applied to deduce

$$f \mid g \overset{t,[m/x]}{\rightarrow} [m/x].f^t \mid g^t.$$

Introducing substitution events allows us to distinguish between $\mathbf{0}$ and $let(x)$. Both $\mathbf{0}$ and $let(x)$ have transitions due to (Subst), e.g., with event $(0, [1/x])$. However, $[1/x].\mathbf{0}^0 = \mathbf{0}$ still has no observable transitions, while $[1/x].let(x)^0 = let(1)$ publishes 1.

Example In the example from Section 4, $(Rtimer(3) \gg M(x)) \mid N$ was shown to evaluate to expression $((\mathbf{0} \gg M(x)) \mid M(x)) \mid \mathbf{0}$, which had no further transitions. The rule (Subst) can now be applied, e.g., with event $(0, [1/x])$ to yield expression $((\mathbf{0} \gg M(1)) \mid M(1)) \mid \mathbf{0}$, which can be evaluated further.

Summary of notations A summary of notation used in the sequel is shown in Figure 7.

$f \overset{t,a}{\rightarrow} g$	: f evaluates in one step to g with event a at time t
$f \overset{u}{\Rightarrow} g$	: f evaluates in multiple steps to g with execution u
f^t	: expression f shifted forward in time by t units
u_t, U_t	: execution or trace u (or set U) delayed by t units
$\overline{u}$, $\overline{U}$	: trace of an execution u (or set U)
$[\![f]\!]$	: the set of executions of f
$\langle\!\langle f \rangle\!\rangle$	: $\overline{[\![f]\!]}$, the set of traces of f
$f \sim g$	: $[\![f]\!] = [\![g]\!]$
$f \cong g$	: $\langle\!\langle f \rangle\!\rangle = \langle\!\langle g \rangle\!\rangle$

Fig. 7. Summary of Notation

7. Identities

In this section, we list certain identities over arbitrary expressions (i.e., with or without free variables), some of them similar to the laws of Kleene algebra [16]. Proofs of the identities, using strong bisimulation, are given in the technical report [27].

(i) $f \mid \mathbf{0} \sim f$
(ii) $f \mid g \sim g \mid f$

(iii) $f \mid (g \mid h) \sim (f \mid g) \mid h$

(iv) $f > x > (g > y > h) \sim (f > x > g) > y > h$, if x is not free in h

(v) $\mathbf{0} > x > f \sim \mathbf{0}$

(vi) $(f \mid g) > x > h \sim f > x > h \mid g > x > h$

(vii) $(f \mid g) < x < h \sim (f < x < h) \mid g$, if x is not free in g

(viii) $(f > y > g) < x < h \sim (f < x < h) > y > g$, if x is not free in g

(ix) $(f < x < g) < y < h \sim (f < y < h) < x < g$,
if y is not free in g and x is not free in h

(x) $\mathbf{0} < x < b \sim b \gg \mathbf{0}$, where b is a site call or handle

Example Continuing the example from Section 6, it is easy to show with the above identities that

$$((\mathbf{0} \gg M(1)) \mid M(1)) \mid \mathbf{0} \cong M(1).$$

8. Denotational Semantics

We propose a denotational semantics of Orc in this section. The denotation of an expression is a set of traces. We show that the denotation of an expression is determined by the denotations of its subexpressions. Thus, the denotational semantics is compositional. Further, we establish in Section 9 that the denotation of expression f is exactly $\langle\!\langle f \rangle\!\rangle$, the trace set of f.

In Sections 8.1, 8.2 and 8.3, we overload the Orc combinators $\mid$, $>x>$ and $<x<$. For any of the three combinators $*$, we define a function $U * V$, where U, V and $U * V$ are sets of executions or traces. These functions are then used in Section 8.4 to formally define the denotations of Orc expressions.

We show later, that each Orc combinator is intimately related to its overloaded counterpart. The following lemma, proved in the technical report [27], illustrates this connection:

Lemma 1 $\langle\!\langle f * g \rangle\!\rangle = \overline{\langle\!\langle f \rangle\!\rangle * \langle\!\langle g \rangle\!\rangle}$

An easy corollary of the above lemma is that the weak bisimulation relation $\cong$ is a congruence.

Corollary 1 *If $f \cong g$, then 1) $f * h \cong g * h$, 2) $h * f \cong h * g$, and 3) $E(p) \cong F(p)$, where $E(x) \ \underline{\Delta}\ f$ and $F(x) \ \underline{\Delta}\ g$.*

Proof. We show only $f * h \cong g * h$; the other proofs are similar.

$$\begin{array}{ll} & \langle\!\langle f * h \rangle\!\rangle \\ = & \{\text{Lemma 1}\} \\ & \overline{\langle\!\langle f \rangle\!\rangle * \langle\!\langle h \rangle\!\rangle} \\ = & \{f \cong g \text{ iff } \langle\!\langle f \rangle\!\rangle = \langle\!\langle g \rangle\!\rangle\} \\ & \overline{\langle\!\langle g \rangle\!\rangle * \langle\!\langle h \rangle\!\rangle} \\ = & \{\text{Lemma 1}\} \\ & \langle\!\langle g * h \rangle\!\rangle \end{array}$$

Note: The definitions of the overloaded operators given in the following three subsections are quite technical. They may be skipped on a first reading.

8.1. *Symmetric Composition*

Our goal in this section is to define the operator $\mid$ over sets of sequences such that $\llbracket f \rrbracket \mid \llbracket g \rrbracket = \llbracket f \mid g \rrbracket$. We first define $u \mid v$, the *partial merge* of sequences u and v, and then lift the definition to sets of sequences.

A merge is an interleaving of events in non-decreasing order of time, in which a substitution event occurs iff if it appears identically in both u and v. The partial merge $u \mid v$ is a merge that only includes events for the range of time that is fully specified in both u and v. The valid time range is from 0 to

$\min(u.time, v.time)$, inclusive. For example, given $u = (0, a)$ and $v = (2, b)$, where a is a base event, the time bound is $\min(u.time, v.time) = 0$ and the partial merge is $p = (0, a)$. For expressions f and g with executions $u \in [\![f]\!]$ and $v \in [\![g]\!]$ where $u = (0, a)$ and $v = (2, b)$, the partial merge of u and v does not include $(0, a)(2, b)$. To see why, suppose that every execution of f extends u with event $(1, c)$. Then $(0, a)(2, b)$ is not a possible execution of $f \mid g$, because it asserts that f need not engage in any event after $(0, a)$ until time 2. The execution u has information about what events must occur up until $u.time$ (the time of the last event of u), but not what must occur after $u.time$. Similarly, an execution has information about what *does not happen*. The execution $(0, a)(2, b)$ specifies that no event occurs between times 0 and 2.

Additional notation is useful for the formal definition of partial merge. Let p be a proposition and S a set. A *guarded set* $[p \;\longrightarrow\; S]$ is defined by

$$[p \;\longrightarrow\; S] = \begin{cases} S & \text{if } p \\ \{\epsilon\} & \text{otherwise} \end{cases}$$

Some additional relations on events are also convenient. Define $a \simeq b$ to mean that a and b are identical substitution events, and $a \preceq b$ to mean that a is a base event and $a.time \leq b.time$. Partial merge is defined by the following rules:

$$\begin{aligned} \epsilon \mid v &= \{\epsilon\} \\ u \mid \epsilon &= \{\epsilon\} \\ au \mid bv &= \quad [a \simeq b \rightarrow a(u \mid v)] \\ &\quad \cup \; [a \preceq b \rightarrow a(u \mid bv)] \\ &\quad \cup \; [b \preceq a \rightarrow b(au \mid v)] \end{aligned}$$

These rules define how events in u and v are merged to produce the executions of the set $u \mid v$. In the first two cases, if either u or v is an empty execution, then the events in the other execution are discarded. The third case applies when both u and v contain at least one event. The result is a union of the different ways in which the events in u and v can be interleaved. If the initial events are the same substitution at the same time, $a \simeq b$, then they are merged. Otherwise the first event in time order is output followed by the merge of the rest of the execution, including the other event. An event a will only be included if there is a corresponding event b at an equal or later time.

Example Consider $u = (0, a)$ and $v = (0, b)$, where a and b are base events. Then $(0, a)(0, b)$ and $(0, b)(0, a)$ are possible merges, because events that occur in the same instant may appear in either order. If $u = (0, a)(2, c)(5, [m/x])$ and $v = (1, b)(5, [m/x])$, then the only merge is $(0, a)(1, b)(2, c)(5, [m/x])$. Time order is preserved and matching substitution events occur only once in the merge.

The definition of $u \mid v$ is lifted pointwise to apply to sets of executions:

$$U \mid V = (\cup u, v : u \in U, v \in V : u \mid v)$$

Full Merge Some of the other semantics functions require a full merge, which includes all the events, not just a prefix. The *full merge* $u + v$ of executions u and v is similar to partial merge but includes all events of u and v. For example, if $u = (0, a)(2, c)$ and $v = (1, b)$, then $(0, a)(1, b)(2, c)$ is a full merge $u + v$, whereas the prefix $(0, a)(1, b)$ is a partial merge $u \mid v$.

8.2. *Sequential Composition*

Our goal in this section is to define the operator $>x>$ over sets of sequences such that $[\![f]\!] >x> [\![g]\!] = [\![f >x> g]\!]$. We first define $u >x> V$, for sequence u and set V, and later lift the definition to $U >x> V$, for set U.

First, define the operator $V\backslash[m/x]$ for a set of sequences V, which informally corresponds to the application of $[m/x]$ to the executions of V. Formally,

$$V\backslash[m/x] = \{v' \mid v \in V \text{ and } v = (0,[m/x])v'\}.$$

The definition of $u \; {>}x{>} \; V$ is given by:

$$\begin{array}{rcrcll}
\epsilon & {>}x{>} & \emptyset & = & \emptyset, & \\
\epsilon & {>}x{>} & V & = & \{\epsilon\} & \text{if } V \neq \emptyset \\
(t,\tau)p & {>}x{>} & V & = & (t,\tau)(p \; {>}x{>} \; V) & \\
(t,[m/x])p & {>}x{>} & V & = & (t,[m/x])(p \; {>}x{>} \; V) & \\
(t,[m/y])p & {>}x{>} & V & = & (t,[m/y])(p \; {>}x{>} \; V\backslash[m/y]) & \text{if } x \neq y \\
(t,!m)p & {>}x{>} & V & = & (t,\tau)(p \; {>}x{>} \; V \mid (V\backslash[m/x])_t) &
\end{array}$$

The first two rules cover the cases when u is ϵ and the remaining rules cover the cases where u has an initial event. If the initial event is τ or a substitution to the bound variable x, then V is not affected and the event is output from the composed expression. If the event is a substitution to a variable other than the bound variable x, then the substitution is output from the composed expression and is also applied to V to create $V\backslash[m/y]$. The final case is the interesting one. If u publishes a value b at time t, then the composite process has an internal τ transition at time t. In addition, a copy of V is created in parallel that receives the substitution $[m/x]$ for its bound variable.

The definition of $u \; {>}x{>} \; V$ is lifted pointwise to apply to sets of executions:

$$U \; {>}x{>} \; V = (\cup u : u \in U : u \; {>}x{>} \; V)$$

8.3. *Asymmetric Composition*

Our goal in this section is to define the operator ${<}x{<}$ over sets of sequences such that $[\![f]\!] \; {<}x{<} \; [\![g]\!] = [\![f \; {<}x{<} \; g]\!]$. We first define $u \; {<}x{<} \; v$, for sequences u and v, and later lift the definition to $U \; {>}x{>} \; V$, for sets U and V.

The semantics of asymmetric composition $u \; {<}x{<} \; v$ is complex, in that it supports parallel execution, communication via the bound variable x, and termination of v. Although u and v are executed in parallel before v publishes, the existing parallel composition operator, partial merge, cannot be used directly because a substitution to a free occurrence of x in v must not be applied to the bound uses of x in u.

The *bound partial merge* operator $u \mid_x v$ is an alternative merge operator that treats x as bound in u. To easily describe substitutions to free or bound variables, we use the term *own-substitution* for a substitution to x and *other-substitution* for a substitution to any variable other than x.

Let $a \approx_x b$ mean that a and b are identical other-substitutions. Let $b \; \underline{\prec\!\!\prec}_x \; a$ mean that (1) b is either a base event or an own-substitution, and (2) $b.time \leq a.time$. The bound partial merge $u \mid_x v$ of u and v is then:

$$\begin{array}{rcl}
\epsilon \mid_x v & = & \{\epsilon\} \\
u \mid_x \epsilon & = & \{\epsilon\} \\
au \mid_x bv & = & \quad [a \approx_x b \rightarrow a(u \mid_x v)] \\
 & & \cup \; [a \preceq b \rightarrow a(u \mid_x bv)] \\
 & & \cup \; [b \; \underline{\prec\!\!\prec}_x \; a \rightarrow b(au \mid_x v)]
\end{array}$$

For proposition p and set S, a *full guarded set* $\langle p \;\longrightarrow\; S \rangle$ is a guarded set, except that $\langle \mathit{false} \;\longrightarrow\; S \rangle = \emptyset$, whereas $[\mathit{false} \;\longrightarrow\; S] = \{\epsilon\}$. Full guarded sets are used to define bound full merge $+_x$:

$$\begin{aligned} u \underset{x}{+} \epsilon &= \begin{cases} \{u\} & \text{if } u \text{ contains no substitution event} \\ \emptyset & \text{otherwise} \end{cases} \\ \epsilon \underset{x}{+} v &= \begin{cases} \{v\} & \text{if } v \text{ contains no other-substitution} \\ \emptyset & \text{otherwise} \end{cases} \\ au \underset{x}{+} bv &= \quad \langle a \approx_x b \rightarrow a(u +_x v)\rangle \\ &\quad \cup \ \langle a \preceq b \rightarrow a(u +_x bv)\rangle \\ &\quad \cup \ \langle b \,\underline{\prec\!\!\prec}_x\, a \rightarrow b(au +_x v)\rangle \end{aligned}$$

Next, define the following conditions:
$d_1(u, v) \equiv$ u has no own-substitution and v has no publication
$d_0(u, v) \equiv$ u and v have the same sequence of other-substitutions

The semantics of asymmetric composition $u <x< v$ can now be defined formally by the following rules:

$$\begin{array}{rcll} u & <x< & v & = u \,|_x\, v & \text{if } d_1(u, v) \\ u'(t, m/x)u'' & <x< & v'(t, !m)v'' & = (u' +_x v')(t, \tau)u'' & \text{if } d_1(u', v') \text{ and } d_0(u', v') \\ u & <x< & v & = \emptyset & \text{otherwise} \end{array}$$

A bound partial merge is used in the first case, when v does not publish a value. The second case, when v publishes, uses a bound full merge for the events up to the substitution to x by u and the publication by v. For u' and v' that satisfy $d_1(u', v')$ and $d_0(u', v')$, *all* events of u' and v' can be included in the merge because the events that follow in u and v are $(t, [m/x])$ and $(t, !m)$, which both occur at t.

Condition $d_1(u, v)$ separates the two main cases: either v does not publish, or it does. If v does not publish, then u must not have an own-substitution, which corresponds to receiving the published value. In the second case, where $v = v'(t, !m)v''$ and $u = u'(t, [m/x])u''$, the prefix v' must not publish. Thus the conditions for all events up to the first publication (if any) of v are the same. Condition d_0 ensures that the merge of u' and v' is well-defined.

Example Let $u = (2, a)(5, [m/x])(7, b)$ and $v = (0, c)(5, !m)$. Here, $u' = (2, a)$ and $v' = (0, c)$. The bound full merge of $u' +_x v'$ is $(0, c)(2, a)$ and the asymmetric composition $u <x< v$ is $(0, c)(2, a)(5, \tau)(7, b)$.

The definition of $u <x< v$ is lifted pointwise to apply to sets of executions:

$$U <x< V = (\cup u, v : u \in U, v \in V : u <x< v)$$

8.4. *Denotation of an Expression*

The goal of this section is to show how to combine the denotations of the subexpressions of f to obtain the denotation of f. Expression f may be a base expression; it may be $g * h$ where g and h are expressions and $*$ is any Orc combinator; or it may be $E(p)$ where $E(x)$ is a defined expression in which the formal parameter x has been replaced by actual parameter p. For each case, we provide a systematic procedure for construction of trace sets (i.e., denotations). Since an expression may be recursively defined, the denotation is defined as the least upper bound of an infinite set of trace sets. Intuitively $\mu_i(f)$, defined below, is the trace set of f in which recursively defined subexpressions have been unfolded i times.

Let A be the set of all finite sequences of substitution events at time 0. Set A will be a subset of the denotation of every Orc expression. Next, define $\mu_i(f)$, for any expression f and for all i, where $i \geq 0$, and $\mu(f)$ as the union over all $\mu_i(f)$:

$$
\begin{aligned}
\mu_0(f) &= A \\
\mu_{i+1}(f) &= \begin{cases} \langle\!\langle b \rangle\!\rangle & \text{if } f = b \text{, a base expression} \\ \overline{\mu_{i+1}(g) * \mu_{i+1}(h)} & \text{if } f = g * h \\ \mu_i([p/x].g) & \text{if } f = E(p) \text{ and } E(x) \;\; \underline{\Delta} \;\; g \end{cases} \\
\mu(f) &= (\cup i : i \geq 0 : \mu_i(f))
\end{aligned}
$$

The denotation of expression f is defined to be $\mu(f)$. In the next section, we show a number of properties of the denotational semantics and its relation to the operational semantics, in particular that $\mu(f)$ is exactly the trace set of f, $\langle\!\langle f \rangle\!\rangle$.

Example Consider the following defined expression App, which repeatedly calls site M using the value from the last publication as input to the next call, and publishes the intermediate value when each call returns.

$$App(x) \;\; \underline{\Delta} \;\; M(x) > y > (let(y) \mid App(y))$$

Then the denotation of $App(m)$ is $\mu(App(m)) = (\cup i : i \geq 0 : \mu_i(App(m)))$. In the example below, we compute $\mu_2(App(m))$.

$$
\begin{array}{ll}
 & \mu_2(App(m)) \\
= & \{\text{definition of } \mu_{i+1}(E(p))\} \\
 & \mu_1([m/x].(M(x) > y > (let(y) \mid App(y))) \\
= & \{\text{definition of substitution}\} \\
 & \mu_1(M(m) > y > (let(y) \mid App(y))) \\
= & \{\text{definition of } \mu_{i+1}(f > x > g)\} \\
 & \overline{\mu_1(M(m)) > y > \mu_1(let(y) \mid App(y))} \\
= & \{\text{definition of } \mu_{i+1}(b) \text{, where } b \text{ is a base expression}\} \\
 & \overline{\langle\!\langle M(m) \rangle\!\rangle > y > \mu_1(let(y) \mid App(y))} \\
= & \{\text{definition of } \mu_{i+1}(f \mid g)\} \\
 & \overline{\langle\!\langle M(m) \rangle\!\rangle > y > \overline{\mu_1(let(y)) \mid \mu_1(App(y))}} \\
= & \{\text{definition of } \mu_{i+1}(b) \text{, where } b \text{ is a base expression}\} \\
 & \overline{\langle\!\langle M(m) \rangle\!\rangle > y > \overline{\langle\!\langle let(y) \rangle\!\rangle \mid \mu_1(App(y))}} \\
= & \{\text{definition of } \mu_{i+1}(E(p))\} \\
 & \overline{\langle\!\langle M(m) \rangle\!\rangle > y > \overline{\langle\!\langle let(y) \rangle\!\rangle \mid \mu_0([y/x].(M(x) > y > (let(y) \mid App(y))))}} \\
= & \{\text{definition of } \mu_0(f)\} \\
 & \overline{\langle\!\langle M(m) \rangle\!\rangle > y > \overline{\langle\!\langle let(y) \rangle\!\rangle \mid A}}
\end{array}
$$

9. Equivalence of the Semantics

Section 4 contains an operational semantics of Orc which allows us to define the set of traces, $\langle\!\langle f \rangle\!\rangle$, of expression f. Section 8 contains a denotational semantics in which we gave $\mu(f)$ as the denotation of f. In this section, we show that the two semantics are equivalent, i.e., $\langle\!\langle f \rangle\!\rangle = \mu(f)$. This result shows that we can reason about the behavior of an Orc program either operationally (using the semantics from Section 4), or using μ, which is both compositional and inductive, and thus, allows a full treatment of recursively defined expressions.

The proof of equivalence of the semantics makes use of the following lemmas, which are proved in the technical report, Wehrman et. al [27]. Notation is summarized in Figure 7, page 7.

Lemma 2 $\mu(f * g) = \overline{\mu(f) * \mu(g)}$
Lemma 3 $\mu(E(p)) = \mu([p/x].g)$, *where* $E(x) \;\; \underline{\Delta} \;\; g$
Lemma 4 $\mu([m/x].f) \subseteq \mu(f)\backslash[m/x]$, (recall that $\mu(f)\backslash[m/x] = \{u \mid (0, [m/x])u \in \mu(f)\}$).

Theorem 1 *(Equivalence of Semantics)* $\langle\!\langle f \rangle\!\rangle = \mu(f)$

Proof. The proof is by induction on both the expression subterm ordering and the usual ordering on the natural numbers.

- $f = b$, a base expression

$$
\begin{array}{ll}
 & \mu(b) \\
= & \{\text{definition of } \mu\} \\
 & A \cup (\cup i : i \geq 0 : \mu_{i+1}(b)) \\
= & \{\text{definition of } \mu_{i+1}(b)\} \\
 & A \cup (\cup i : i \geq 0 : \langle\!\langle b \rangle\!\rangle) \\
= & \{A \subseteq \langle\!\langle f \rangle\!\rangle, \text{ for any expression } f\} \\
 & \langle\!\langle b \rangle\!\rangle.
\end{array}
$$

- $f = g * h$:

$$
\begin{array}{ll}
 & \langle\!\langle g * h \rangle\!\rangle \\
= & \{\text{Lemma 1}\} \\
 & \overline{\langle\!\langle g \rangle\!\rangle * \langle\!\langle h \rangle\!\rangle} \\
= & \{\text{induction}\} \\
 & \overline{\mu(g) * \mu(h)} \\
= & \{\text{Lemma 2}\} \\
 & \mu(g * h)
\end{array}
$$

- $f = E(p)$, where $E(x) \;\underline{\Delta}\; g$. The proof is by mutual inclusion.

  ○ $\mu(E(p)) \subseteq \langle\!\langle E(p) \rangle\!\rangle$: We show, for all $i \geq 0$, that $\mu_i(E(p)) \subseteq \langle\!\langle E(p) \rangle\!\rangle$. We proceed by induction on i.

  For $i = 0$, $\mu_0(E(p)) = A$, and $A \subseteq \langle\!\langle E(p) \rangle\!\rangle$ by definition. Now, we assume that $\mu_i(E(p)) \subseteq \langle\!\langle E(p) \rangle\!\rangle$ and show that $\mu_{i+1}(E(p)) \subseteq \langle\!\langle E(p) \rangle\!\rangle$. Let $u \in \mu_{i+1}(E(p))$.

$$
\begin{array}{ll}
 & u \in \mu_{i+1}(E(p)) \\
\Rightarrow & \{\text{definition}\} \\
 & u \in \mu_i([p/x].g) \\
\Rightarrow & \{\text{induction on } i\} \\
 & u \in \langle\!\langle [p/x].g \rangle\!\rangle \\
\Rightarrow & \{\text{by definition, for some } v \text{ such that } \overline{v} = u\} \\
 & v \in [\![\, [p/x].g \,]\!] \\
\Rightarrow & \{\text{from rule (CALL) of operational semantics in Figure 3, } E(x) \;\underline{\Delta}\; g\} \\
 & (0, \tau)v \in [\![\, E(p) \,]\!] \\
\Rightarrow & \{\overline{(0, \tau)v} = \overline{v} = u,\ \overline{[\![\, E(p) \,]\!]} = \langle\!\langle E(p) \rangle\!\rangle\} \\
 & u \in \langle\!\langle E(p) \rangle\!\rangle
\end{array}
$$

  ○ $\langle\!\langle E(p) \rangle\!\rangle \subseteq \mu(E(p))$: Consider $u \in \langle\!\langle (E(p)) \rangle\!\rangle$. Let $v \in [\![\, E(p) \,]\!]$ such that $\overline{v} = u$. We show that $u = \overline{v} \in \mu(E(p))$. The proof proceeds by induction on the length of v.

  It is easy to show by induction on i that $\epsilon \in \mu_i(E(p))$; therefore $\epsilon \in \mu(E(p))$. So let $v = av'_t$, where a is some event at time t. If a is a substitution event, then $t = 0$ because $E(p)^t = \perp$ for $t > 0$.

$$
\begin{array}{ll}
 & av' \in [\![\, E(p) \,]\!] \\
\Rightarrow & \{\text{operational semantics}\} \\
 & v' \in [\![\, a.E(p) \,]\!] \\
\Rightarrow & \{\text{definition of trace}\} \\
 & \overline{v'} \in \langle\!\langle a.E(p) \rangle\!\rangle \\
\Rightarrow & \{\text{induction on the length of } v'\} \\
 & \overline{v'} \in \mu(a.E(p)) \\
\Rightarrow & \{\mu(a.E(p)) \subseteq \mu(E(p)) \backslash a \text{ by Lemma 4, page 15}\} \\
 & \overline{v'} \in \mu(E(p)) \backslash a \\
\Rightarrow & \{\text{definition of } \backslash\}
\end{array}
$$

$\quad a\overline{v'} \in \mu(E(p))$
$\Rightarrow \quad \{a\overline{v'} = \overline{av'}\}$
$\quad \overline{av'} \in \mu(E(p))$

If a is not a substitution event, by rule (Def), $E(p) \overset{0,\tau}{\to} [p/x].g \overset{v'}{\Rightarrow}$, where $E(x) \ \underline{\Delta}\ g$. So $v = (0,\tau)v'$.

$\quad (0,\tau)v' \in [\![\, E(p) \,]\!]$
$\Rightarrow \quad \{\text{operational semantics}\}$
$\quad v' \in [\![\, [p/x].g \,]\!]$
$\Rightarrow \quad \{\text{definition of trace}\}$
$\quad \overline{v'} \in \langle\!\langle [p/x].g \rangle\!\rangle$
$\Rightarrow \quad \{\text{induction on } v'\}$
$\quad \overline{v'} \in \mu([p/x].g)$
$\Rightarrow \quad \{\text{Lemma 3}\}$
$\quad \overline{v'} \in \mu(E(p))$
$\Rightarrow \quad \{\overline{v'} = \overline{(0,\tau)v'}\}$
$\quad \overline{(0,\tau)v'} \in \mu(E(p))$

10. Related Work

There has been extensive work on timed semantics for concurrent languages. The approach to time taken here is similar to previous studies: each event is associated with a time. The differences arise in the way time constraints are specified within the language under study. There are several models of time for Petri nets, including Timed Petri Nets [11] and Time Petri Nets [3]. Time may be associated with tokens, places, or transitions. In some cases time delays are fixed quantities, while other studies allow a finite range of times.

Temporal variants of other process calculi have also been studied. Temporal Process Language (TPL) [12] is a variant of CCS [18] with time. Linear-time π-calculus [25] augments π-calculus [19] with temporal operators. Berger [5] gives a congruence relation for a version of π-calculus with timers [6]. A clock-step function is used to give meaning to timer expressions; so, time is discrete in this model. The approach taken in this paper is similar to that of ACP, a discrete time process algebra, in using a delay operator and silent actions [4], or ATP with idling actions [21]. Linda-like coordination languages with fixed-time relative and absolute delay operators have been studied recently [17,10].

AlTurki and Meseguer [2] have proposed a different timed operational semantics of Orc. In their semantics, which extends the asynchronous semantics, site calls and responses are modeled with a message pool. A clock-tick event is used to update the state of messages in the pool and is restricted to quiescent expressions for which no internal event is enabled. They provide a translation of the semantics to rewriting logic using Maude [8], which provides a certified implementation and an LTL model checker. Because the semantics is based on clock-tick events, time must be modeled discretely. The semantics presented here may be based on either a discrete or continuous notion of time.

Bruni, et al have proposed SCC [7], a service-oriented process calculus inspired by Orc and the π-calculus which includes a mechanism for handling sessions between a client and server. SCC supports bi-directional communication between clients and services using a mechanism for passing channel names among processes. Although such complex protocols can be encoded in Orc (e.g., using by encoding channels as sites), the language features of SCC simplify practical programming. The goal of Orc is to establish a foundation on which practical programming languages can be built.

Vardoulakis and Wand have developed an alternate asynchronous operational and denotational semantics for Orc [26]. Their work addresses an ambiguity in the semantics of [15] regarding the treatment of free variables. They introduce an explicit variable context into the operational semantics, which restricts the occurrence of substitution events. An expression may only transition with a receive event, and undergo the corresponding substitution, when a binding for the substituted variable is available in the context. From this operational semantics, they derive a denotational semantics where denotations are functions from contexts to traces, rather than just traces.

11. Conclusion

The structured concurrency model of Orc lends itself naturally to task orchestrations. Task orchestration is a form of structured concurrent programming in which an agent invokes and coordinates the execution of passive, potentially unreliable services. Orchestration is well-suited to solving a range of concurrency problems, most notably workflow. Most practical applications deal explicitly with time, either to schedule activities or to deal with timeouts and delays. This article develops a timed semantics for Orc in order to provide a simple, well-defined interpretation of Orc in the presence of time. The semantics is shown both operationally and denotationally, where the denotations are traces of events labeled by the time at which they occur. Equivalence of the semantics allows us to reason about Orc programs both operationally and compositionally. The timed semantics enjoys the same properties and identities as the previous asynchronous semantics.

The timed semantics brings Orc closer to its original intended design. In particular, Orc itself is eager, while the environment may cause arbitrary delays. In the semantics, Orc must call sites and publish results as soon as possible, while a remote site, which exists in the environment, may respond with arbitrary delay, or not at all. The semantics also corresponds closely to our prototype implementation of Orc.

References

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

[2] Musab AlTurki and José Meseguer. Real-time Rewriting Semantics of Orc. In Michael Leuschel and Andreas Podelski, editors, *PPDP*, pages 131–142. ACM, 2007.

[3] Tuomas Aura and Johan Lilius. Time Processes for Time Petri-Nets. In *ICATPN*, volume 1248 of *LNCS*, pages 136–155, 1997.

[4] J. C. M. Baeten and J. A. Bergstra. Discrete Time Process Algebra. *Formal Aspects of Computing*, 8:188–208, 1996.

[5] Martin Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous pi-Calculi. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2004.

[6] Martin Berger and Kohei Honda. The Two-Phase Commitment Protocol in an Extended pi-Calculus. *Electr. Notes Theor. Comput. Sci.*, 39(1), 2000.

[7] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. SCC: A Service Centered Calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[8] Christiano Braga, Manuel Clavel, Francisco Durán, Steven Eker, Azadeh Farzan, Joe Hendrix, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Peter Olveczky, Miguel Palomino, Ralf Sasse, Mark-Oliver Stehr, Carolyn Talcott, and Alberto Verdejo. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow Patterns in Orc. In *Proc. of the International Conference on Coordination Models and Languages*, 2006.

[10] Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A Timed Linda Language and its Denotational Semantics. *Fundamenta Informatica*, 63(4):309–330, 2004.

[11] Alois Ferscha. Concurrent Execution of Timed Petri Nets. In *Winter Simulation Conference*, pages 229–236, 1994.

[12] Matthew Hennessy and Tim Regan. A Process Algebra for Timed Systems. *Inf. Comput.*, 117(2):221–239, 1995.

[13] Tony Hoare, Galen Menzel, and Jayadev Misra. A Tree Semantics of an Orchestration Language. In Manfred 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`.

[14] David Kitchin, William R. Cook, and Jayadev Misra. A Language for Task Orchestration and Its Semantic Properties. In *CONCUR*, pages 477–491, 2006.

[15] David Kitchin, William R. Cook, and Jayadev Misra. Semantic Properties of Asynchronous Orc. Technical Report TR-06-32, University of Texas at Austin, Department of Computer Sciences, 2006.

[16] Dexter Kozen. On Kleene Algebras and Closed Semirings. In *Proceedings, Math. Found. of Comput. Sci.*, volume 452 of *LNCS*, pages 26–47. Springer-Verlag, 1990.

[17] Isabelle Linden, Jean-Marie Jacquet, Koen De Bosschere, and Antonio Brogi. On the Expressiveness of Timed Coordination Models. *Sci. Comput. Program.*, 61(2):152–187, 2006.

[18] R. Milner. *Communication and Concurrency.* International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall, 1989.

[19] Robin Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, May 1999.

[20] Jayadev Misra and William R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. *Journal of Software and Systems Modeling*, May, 2006. Available for download at `http://dx.doi.org/10.1007/s10270-006-0012-1`.

[21] X. Nicollin and J. Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[22] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[23] Sydney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Net Systems Semantics of Web Services Orchestrations Modeled in Orc. Technical Report PI 1780, IRISA, 2006.

[24] Sydney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. SLA for Web Services Orchestrations. Unpublished manuscript, 2006.

[25] Colin Stirling. *Modal and Temporal Properties of Processes.* Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[26] Dimitrios Vardoulakis and Mitchell Wand. A Compositional Trace Semantics for Orc. Personal communication, 2007.

[27] Ian Wehrman, David Kitchin, William R. Cook, and Jayadev Misra. Properties of the Timed Operational and Denotational Semantics of Orc. Technical Report TR-07-65, University of Texas at Austin, Department of Computer Sciences, 2007.