A Language for Task Orchestration and its Semantic Properties

David Kitchin, William R. Cook and Jayadev Misra [⋆]

The University of Texas at Austin
{dkitchin,wcook,misra}@cs.utexas.edu

Abstract. Orc is a new language 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 failure of services or communication. In this paper, we show a trace-based semantic model for Orc, which induces a congruence on Orc programs and facilitates reasoning about them. Despite the simplicity of the language and its semantic model, Orc is able to express a variety of useful orchestration tasks.

1 Introduction

We describe the semantic properties of *Orc*, a new language for *task orchestration*. Task orchestration is a form of concurrent programming in which multiple services are invoked to achieve a goal – while managing time-outs, priorities, and failures of services and 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.

Many practical problems can be understood as orchestrations – for example, business workflows are naturally expressed as orchestrations [1]. We illustrate the use of Orc in implementing some traditional concurrent computation patterns; larger examples have also been developed [2, 3]. Orc has also been used to study service-level agreements for composite web services [4].

The goal of this work is to develop a language based on a simple trace semantics that can still express, and support reasoning about, useful task orchestrations. The key metric for a trace semantics is whether trace equivalence corresponds to observational equivalence of programs. Depending on the language, traces must be extended to include failures, refusals, ready states, etc., in order to adequately model observational equivalence. The trace semantics of Orc is a simple set of traces; the traces include communication events and substitution events, which model synchronization. We prove that the equality of trace sets defines a congruence on programs, in that programs with equivalent trace sets are interchangeable. We show a number of laws about Orc programs, similar to those in Kleene algebra [5]; the laws are based on strong bisimulation. We then introduce a more general congruence based on trace set equivalence which can establish laws not provable by strong bisimulation.

[⋆] Work of the second author is partially supported by National Science Foundation grant CCF-0448128.

2 Overview of Orc

An Orc program consists of a set of definitions and a *goal* expression which is to be evaluated. Orc assumes that basic services, like sequential computation and data manipulation, are implemented by primitive *sites*. Orc provides constructs to orchestrate the concurrent invocation of sites.

The syntax of Orc is given in Figure 1. A site call $M(\bar{p})$ invokes a site named M with a list of actual parameters $\bar{p}$. If there are no parameters, the site call is written as just 'M'. An actual parameter p may be a variable x or a value v. Calls to defined expressions $E(\bar{p})$ are similar, given a definition with name E and formal parameters $\bar{p}$. There are only three combinators: $>x>$ for sequential composition, $\mid$ for parallel composition, and **where** for asymmetric parallel composition. The combinators are listed in decreasing order of precedence, so $f > x > g \mid h$ means $(f > x > g) \mid h$, and f **where** $x :\in g \mid h$ means f **where** $x :\in (g \mid h)$. In the remainder of this section, we give an informal overview of the Orc programming model with examples. The formal semantics is given in Section 3.

2.1 Site Calls

The simplest Orc expression is a *site call* $M(\bar{p})$, where M is a site name and $\bar{p}$ is a list of actual parameters. 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. For example, a site call $CNN(d)$, where CNN is a news service and d is a date, might download the newspage for the specified date. Site calls are *strict*, i.e., a site is called only if all its parameters have values.

Table 1 defines a few sites that are fundamental to effective programming in Orc (a *signal* is a value which has no additional information). Additionally, $\mathbf{0}$ represents a site which never responds.

2.2 Combinators

There are three combinators in Orc to compose expressions. Symmetric composition of f and g, written as $f \mid g$, evaluates f and g independently. The sites called

$$\begin{array}{rcll} f, g, h & \in & \textit{Expression} & ::= M(\bar{p}) \;\big|\; E(\bar{p}) \;\big|\; f \mid g \;\big|\; f > x > g \;\big|\; f \textbf{ where } x :\in g \\ p & \in & \textit{Actual} & ::= x \;\big|\; v \\ & & \textit{Definition} & ::= E(\bar{x}) \;\underline{\Delta}\; f \end{array}$$

Fig. 1. Syntax of Orc

$let(x, y, \cdots)$	Returns argument values as a tuple.
$if(b)$	Returns a signal if b is true, and does not respond if b is false.
$Rtimer(t)$	Returns a signal after exactly t time units.

Table 1. Fundamental Sites

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 the number of sites that respond.

In $f > x > g$, expression f is evaluated, each value published by it initiates a fresh instance of g as a separate computation, and the value published by f is called x in g's computation. Evaluation of f continues while (possibly several) instances of g are run. This is the only mechanism in Orc similar to spawning threads. If f publishes no value, g is never instantiated. The values published by the executions of different instances of g are the values published by $f > x > g$. 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 a short-hand for $f > x > g$ when x is not used in g.

To evaluate (f **where** $x :\in g$), 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. In $((M \mid N(x))$ **where** $x :\in R)$, for example, M can be called even before x has a value. If g publishes a value, then x is assigned this value, g's evaluation is terminated and the suspended parts of f can proceed. This is the only mechanism in Orc to block or terminate parts of a computation. Unlike in the previous example, the following expression calls $email$ at most once.

$$email(a, x) \textbf{ where } x :\in (CNN(d) \mid BBC(d))$$

2.3 Definitions

Declaration $E(\bar{x}) \;\underline{\Delta}\; f$ defines expression E whose formal parameter list is $\bar{x}$ and body is expression 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.

2.4 Examples

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

$$let(z) \textbf{ where } z :\in (\ M > x > let(x, true) \ \mid \ Rtimer(t) > x > let(x, false)\ )$$

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. The following code fragment calls sites M and N in parallel and publishes their values as a tuple after they both complete their executions.

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

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.

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

Priority Call the sites M and N, but give priority to M by publishing its response if it arrives within the first time unit, even if N's response precedes it.

$$let(x) \textbf{ where } x :\in (M \mid ((Rtimer(1) \gg let(u)) \textbf{ where } u :\in N))$$

Arbitration A fundamental problem in concurrent computing is *arbitration*: to choose between two computations and let only one proceed. Arbitration is the essence of mutual exclusion. Consider a process which behaves as process P if event $Alpha$ happens and as Q if $Beta$ happens. In Orc, events $Alpha$ and $Beta$ are represented as sites, and P and Q are expressions. Below, $flag$ records which of $Alpha$ and $Beta$ responds first.

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

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

Recursive definitions of expressions The recursive expression *Metronome*, defined below, publishes a signal every time unit, starting immediately. It is used in the subsequent expression to call site *Poll* each time unit and publish its values.

$$\begin{array}{l} \mathit{Metronome} \ \underline{\Delta} \ \mathit{Signal} \mid \mathit{Rtimer}(1) \gg \mathit{Metronome} \\ \mathit{Metronome} \gg \mathit{Poll} \end{array}$$

Non-strict Evaluation; Parallel-or *Parallel-or* is a classic problem in non-strict evaluation: computation of $x \vee y$ over booleans x and y publishes *true* if either variable value is *true*; therefore, the evaluation may terminate even when one of the variable values is unknown. Here, we state the problem in Orc terms, and give a simple solution.

Suppose sites M and N publish booleans. Compute the *parallel-or* of the two booleans, i.e., (in a non-strict fashion) publish *true* as soon as either site returns *true* and *false* only if both sites return *false*. In the following solution, site $or(x, y)$ returns $x \vee y$. Site $\mathit{ift}(b)$ returns *true* if b is true; it does not respond otherwise: $\mathit{ift}(b) = \mathit{if}(b) \gg \mathit{let}(\mathit{true})$.

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

3 Asynchronous Semantics

We develop a formal semantics of Orc in this section. The semantics is operational, asynchronous, and based on labeled transition systems. A synchronous semantics has also been defined [2], while a complete temporal semantics is future work.

As is common in small-step operational semantics, the syntax of Orc must be extended to represent intermediate states. We introduce $?k$ to denote an instance of a site call that has not yet returned a value, where k is a unique handle that identifies the call instance. We also restrict the language to sites and definitions with a single argument; multiple arguments are easily handled by adding tuples to the language.

The transition relation $f \overset{a}{\rightarrow} f'$, defined in Figure 2, states that expression f may transition with event a to expression f'. There are four kinds of events, which we call *base events*:

$$a, b \in \mathit{BaseEvent} ::= !v \ \big| \ \tau \ \big| \ M_k(v) \ \big| \ k?v$$

A *publication* event, $!v$, publishes a value v from an expression. As is traditional, τ denotes an *internal* event. The remaining two events, the *site call* event $M_k(v)$ and the *response* event $k?v$, are discussed below.

$$\frac{k \text{ fresh}}{M(v) \overset{M_k(v)}{\rightarrow} ?k} \ (\textsc{SiteCall}) \qquad \frac{f \overset{a}{\rightarrow} f' \quad a \neq !v}{f >x> g \overset{a}{\rightarrow} f' >x> g} \ (\textsc{Seq1N})$$

$$?k \overset{k?v}{\rightarrow} \mathit{let}(v) \ (\textsc{SiteRet}) \qquad \frac{f \overset{!v}{\rightarrow} f'}{f >x> g \overset{\tau}{\rightarrow} (f' >x> g) \mid [v/x].g} \ (\textsc{Seq1V})$$

$$\mathit{let}(v) \overset{!v}{\rightarrow} \mathbf{0} \ (\textsc{Let})$$

$$\frac{f \overset{a}{\rightarrow} f'}{f \mid g \overset{a}{\rightarrow} f' \mid g} \ (\textsc{Sym1}) \qquad \frac{f \overset{a}{\rightarrow} f'}{f \ \textbf{where} \ x :\in g \overset{a}{\rightarrow} f' \ \textbf{where} \ x :\in g} \ (\textsc{Asym1N})$$

$$\frac{g \overset{a}{\rightarrow} g'}{f \mid g \overset{a}{\rightarrow} f \mid g'} \ (\textsc{Sym2}) \qquad \frac{g \overset{!v}{\rightarrow} g'}{f \ \textbf{where} \ x :\in g \overset{\tau}{\rightarrow} [v/x].f} \ (\textsc{Asym1V})$$

$$\frac{(E(x) \ \underline{\Delta} \ f) \in D}{E(p) \overset{\tau}{\rightarrow} [p/x].f} \ (\textsc{Def}) \qquad \frac{g \overset{a}{\rightarrow} g' \quad a \neq !v}{f \ \textbf{where} \ x :\in g \overset{a}{\rightarrow} f \ \textbf{where} \ x :\in g'} \ (\textsc{Asym2})$$

Fig. 2. Asynchronous Operational Semantics of Orc

3.1 Site Calls

A site call involves three steps: invocation of the site, response from the site, and publication of the result. These steps can be arbitrarily interleaved with other site calls, or delayed indefinitely.

Rule SiteCall specifies that a site call $M(v)$, where v is a value, transitions to $?k$ with event $M_k(v)$. The handle k connects a site call to a site return – a fresh handle is created for each call to identify that call instance. The resulting expression, $?k$, represents a process that is blocked waiting for the response from the call. A site call occurs only when its parameters are values; in $M(x)$, where x is a variable, the call is blocked until x is defined.

In SiteRet a pending site call $?k$ receives a result v from the environment and transitions to the expression $\mathit{let}(v)$. There is no assumption that all site calls eventually respond. If there is no response, then the call blocks indefinitely.

The Let rule generates a publication event $!v$ from its argument value v.

3.2 Composition Rules

The composition rules are straightforward, except in some cases where subexpressions publish values. When f publishes a value ($f \overset{!v}{\rightarrow} f'$), rule Seq1V creates a new instance of the right side, $[v/x].g$, the expression in which all free occurrences of x in g are replaced by v. The publication $!v$ is hidden, and the entire expression performs a τ action. Note that f and all instances of g are executed in parallel. Because the semantics is asynchronous, there is no guarantee that the values published by the first instance will precede the values of later instances. Instead, the values produced by all instances of g are interleaved arbitrarily.

Asymmetric parallel composition is similar to parallel composition, except when g publishes a value v. In this case, rule ASYM1V terminates g and x is bound to v 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 D.

4 Executions and Traces

Define the relation $\Rightarrow$ as the transitive closure of the transition relation $\rightarrow$, together with the empty transition ϵ. If $f \overset{s}{\Rightarrow} f'$, we say that s is an *execution* of f.

$$f \overset{\epsilon}{\Rightarrow} f \qquad\qquad \frac{f \xrightarrow{a} f'',\ f'' \overset{s}{\Rightarrow} f'}{f \overset{as}{\Rightarrow} f'}$$

We write as to denote the concatenation of event a onto execution s. Similarly, st is the concatenation of two executions s and t. We have included $f \overset{\epsilon}{\Rightarrow} f$ to guarantee that the set of executions of an expression is prefix-closed.

A *trace* is obtained by removing all internal events, τ, from an execution. Note that every execution (and trace) is finite in length. However, we can represent an infinite sequence of transitions by the set of all of its finite prefixes.

Example An execution of $((M(x) \mid let(x)) >y> R(y))$ **where** $x :\in (N \mid S)$ is shown below. In the following, N returns value 5 and $R(5)$ returns 7.

$$S_k \quad N_l \quad l?5 \quad M_m(5) \quad \tau \quad R_n(5) \quad n?7 \quad !7$$

The response from S, if any, is ignored after N responds. The event τ is due to $let(5) \xrightarrow{!5} \mathbf{0}$. This event causes the event $R_n(5)$. Site call $M_m(5)$ has not yet responded in this execution. The final expression is $(?m >y> R(y))$.

4.1 Laws proved using Strong Bisimulation

A *closed* expression is one which has no free variables; an *open* expression has free variables. In this section, we list certain identities over closed expressions, some of them similar to the laws of Kleene algebra [5]. We write $f \sim g$ to denote that f and g are strongly bisimilar [6]. In a later section, we develop a notion of congruence over both closed and open expressions. That notion can be used to show, for example, that $f >x> let(x)$ is congruent to f.

Below, "f is x-free" means that x is not a free variable of f.

1. $f \mid \mathbf{0} \sim f$
2. $f \mid g \sim g \mid f$
3. $f \mid (g \mid h) \sim (f \mid g) \mid h$
4. $f >x> (g >y> h) \sim (f >x> g) >y> h$, if h is x-free.
5. $\mathbf{0} >x> f \sim \mathbf{0}$
6. $(f \mid g) >x> h \sim f >x> h \mid g >x> h$
7. $(f \mid g)$ **where** $x :\in h \sim (f$ **where** $x :\in h) \mid g$, if g is x-free.
8. $(f >y> g)$ **where** $x :\in h \sim (f$ **where** $x :\in h) >y> g$, if g is x-free.
9. $(f$ **where** $x :\in g)$ **where** $y :\in h \sim (f$ **where** $y :\in h)$ **where** $x :\in g$, if g is y-free and h is x-free.
10. $\mathbf{0}$ **where** $x :\in ?k \sim ?k \gg \mathbf{0}$
11. $\mathbf{0}$ **where** $x :\in M \sim M \gg \mathbf{0}$, for any site M.

4.2 Substitution Events

Strong bisimulation is applicable only if each side of an identity is a closed expression. If we relax this restriction, we can use bisimulation to prove, for example, that $\mathbf{0} = let(x)$, because neither has a non-trivial transition. Yet, these two expressions display different behaviors in the same context. For example, $let(1) >x> \mathbf{0}$ never publishes whereas $let(1) >x> let(x)$ always publishes.

To obtain a more general theory we introduce a new kind of event, a *substitution event* of the form $[v/x]$, and the transition rule:

$$f \xrightarrow{[v/x]} [v/x].f \qquad \text{(SUBST)}$$

A substitution event differs from the base events described in Section 3 in a crucial way: the rules in Figure 2 are defined only over base events. Therefore, given that $f \xrightarrow{[v/x]} [v/x].f$, (SYM1) can *not* be applied to deduce

$$f \mid g \xrightarrow{[v/x]} [v/x].f \mid g,$$

Allowing substitution events expands the set of executions (and traces) of expressions. For example, the traces of $let(x)$ are of the form $[v/x]\ !v$, for all possible v, and their prefixes. Introducing substitution events allows us to distinguish between $\mathbf{0}$ and $let(x)$, for instance, because a possible trace of $let(x)$ is $[1/x]\ !1$.

We observe that proofs by strong bisimulation of the laws of Section 4.1 remain valid after allowing for substitution events. This is because both sides of an identity $f \sim g$ are closed. Hence, given $f \xrightarrow{a} f'$, either a is not a substitution event, or it is of the form $[v/x]$ where x is not free in f. In the latter case, $f' = f$, and this transition corresponds in g to $g \xrightarrow{a} g$.

Furthermore, we can now show that all the laws of Section 4.1 hold for arbitrary expressions f, g, and h, open or closed. So, we can prove, for instance, that $let(x) \mid let(y) \sim let(y) \mid let(x)$. Each side of an identity has the same set of free variables (except $\mathbf{0} >x> f \sim \mathbf{0}$, which is easily handled). Therefore, a proof by strong bisimulation applies to each identity.

Variable Naming Free and bound variables of an expression have different names. Hence, in $(f >x> g)$, f does not have a free variable x, and in $(f$ **where** $x :\in g)$, x is not free in g.

5 Characterizations of Traces

Notation We write $\langle f \rangle$ for the set of traces of f.

In this section, we show that the traces of an expression can be determined from the traces of its constituent subexpressions. In particular, we overload the Orc combinators to apply over trace sets and prove that

$$\begin{aligned}&\langle f \mid g \rangle = \langle f \rangle \mid \langle g \rangle \\ &\langle f >x> g \rangle = \langle f \rangle >x> \langle g \rangle \\ &\langle f \textbf{ where } x :\in g \rangle = \langle f \rangle \textbf{ where } x :\in \langle g \rangle\end{aligned}$$

5.1 Trace characterization of base expressions

Notation We use the following notation for quantified expressions: $(\cup r : r \in R : e)$ denotes $\cup_{r \in R}(e)$, where variable r can be free in e. Range R of r may be omitted if it is clear from context, e.g., $(\cup i :: S_i)$.

We show the trace sets of base expressions, i.e, those without constituent subexpressions. We only list the compact versions of traces in which there are no substitutions to irrelevant variables (which have no effect on the rest of the trace). Also, we only list the maximal traces below, whose prefixes constitute the entire trace set. Below, *Values* denotes the set of all possible responses from sites.

$$\begin{aligned}&\langle let(v) \rangle = \{\ !v\} \\ &\langle M(v) \rangle = (\cup w : w \in \mathit{Values} : \{M_k(v)\ k?w\ \ !w\}) \\ &\langle M(x) \rangle = (\cup v : v \in \mathit{Values} : (\cup t : t \in \langle M(v) \rangle : \{[v/x]t\}))\end{aligned}$$

The trace set for $let(v)$ is easy to see. For $M(v)$, any maximal trace is of the form $M_k(v)\ \ k?w\ \ !w$, where w is a response from M. Note that k is a bound parameter of the trace (with $M_k(v)$ as its binder) and it can be renamed consistently. The trace set of $M(x)$ starts with a substitution $[v/x]$, for any v, followed by any trace of $M(v)$.

5.2 Trace characterization for $(f \mid g)$

Separation and Merge Let s, t and p be sequences of events. We call p a *merge* of s and t if (1) s and t are both subsequences of p and every event of p belongs to at least one of s and t, (2) every common event of p (i.e., an event that belongs to both s and t) is a substitution, and (3) for any variable which has a substitution in both s and t, its first substitution in both s and t is a common event of p. We call the pair $(s,\ t)$ a *separation* of p.

Example Let

$$\begin{aligned}s &= a \quad b \quad [3/x] \quad [4/x] \quad [5/x] \\ t &= c \quad [2/y] \quad [3/x] \quad [5/x] \quad [4/x] \\ u &= [3/x] \quad [2/y]\end{aligned}$$

Below, we use subscripts on events to identify the sequences to which they belong, when there is ambiguity.

$$\begin{array}{llllllll} a \quad c \quad b & [2/y] & [3/x]_{s,t} & [4/x]_s & [5/x]_{s,t} & [4/x]_t & \in (s \mid t) \\ a \quad b & [3/x]_{s,u} & [4/x] & [5/x] & [2/y] & & \in (s \mid u)\end{array}$$

There is no merge for t and u because the orders of first substitutions to x and y are different. Also, if two sequences have $[v/x]$ and $[w/x]$ as their first substitutions for x, and $v \neq w$, then they have no merge, from condition (3). Note that in the merge of s and t, $[5/x]$ appears once, whereas $[4/x]$ appears twice. Condition (3) imposes a constraint only on the first substitution to a variable; subsequent substitutions may or may not be common events in a merge.

Definition For traces s and t, define $s \mid t$ to be the set of their merges.

$$s \mid t = \{p \mid p \text{ is a merge of } s \text{ and } t\}$$

We lift the definition to trace sets $S \mid T$:

$$S \mid T = \{p \mid p \in s \mid t,\ s \in S,\ t \in T\}$$

Note that $s \mid \epsilon = \{s\}$, $S \mid \{\epsilon\} = S$, and $\mid$ over traces is commutative.

Theorem 1 $\langle f \mid g \rangle = \langle f \rangle \mid \langle g \rangle$

Proof Sketch: The complete proof is in [7]. The proof is in two parts showing that one side is a subset of the other. For $\langle f \mid g \rangle \subseteq \langle f \rangle \mid \langle g \rangle$, we show a separation (s, t) of any trace p of $\langle f \mid g \rangle$ such that $s \in \langle f \rangle$ and $t \in \langle g \rangle$. The proof is by induction on the length of p. In the other direction, to show $\langle f \rangle \mid \langle g \rangle \subseteq \langle f \mid g \rangle$, let p be a trace with separation (s,t) where $s \in \langle f \rangle$ and $t \in \langle g \rangle$. We prove that $p \in \langle f \mid g \rangle$ by induction on the length of p.

5.3 Trace characterization for $(f >x> g)$

Define operator $\backslash$ as follows: $T\backslash[v/x] = \{t \mid [v/x]t \in T\}$. That is, $T\backslash[v/x]$ discards all traces in T that do not begin with $[v/x]$, and removes the leading $[v/x]$ event from the remaining traces.

We extend this notation to sequences of substitutions:

$$\begin{aligned}&T\backslash\epsilon = T \\ &T\backslash(cD) = (T\backslash c)\backslash D,\end{aligned}$$

where c is a substitution and D a sequence of substitutions.

Definition For trace s and trace set T define a set of traces $(s >x> T)$ by

$$\begin{cases} \{s\} & \text{if } s \text{ has no publication,} \\ r(t >x> T' \mid (T'\backslash[v/x])) & \text{if } s = r\ \ !v\ t \text{ and } r \text{ has no publication,} \\ \quad \text{where } D \text{ is the sequence of substitutions in } r, & \\ \quad \text{and } T' = T\backslash D & \end{cases}$$

We lift the definition to $S >x> T$, where S and T are sets of traces.

$$S >x> T = \{p \mid p \in s >x> T,\ s \in S\}$$

Theorem 2 $\langle f >x> g\rangle = \langle f\rangle >x> \langle g\rangle$

Proof Sketch: The complete proof is in [7]. The proof is in two parts showing that one side is a subset of the other. To prove that $\langle f\rangle >x> \langle g\rangle \subseteq \langle f >x> g\rangle$, take any p which is in $\langle f\rangle >x> \langle g\rangle$, i.e., $p \in (s >x> \langle g\rangle)$, where $s \in \langle f\rangle$. Then show, by induction on the number of publications in s, that $p \in \langle f >x> g\rangle$. In the other direction, to show $\langle f >x> g\rangle \subseteq \langle f\rangle >x> \langle g\rangle$, we take a trace p of $\langle f >x> g\rangle$, and construct a sequence s which corresponds to the subsequence of events from f in p. We prove, by induction on the number of publications in s, that $p \in (s >x> \langle g\rangle)$.
Note: Any substitution event $[v/x]$ in s is unrelated to x in $(s >x> T)$.

5.4 Trace characterization for $(f$ **where** $x :\in g)$

Definition For traces s and t define a set of traces $(s\ \textbf{where}\ x :\in t)$ by

$$\begin{cases} (s \mid t) & \text{if } t \text{ has no publication,} \\ (s' \mid t')s'' & \text{if } s = s'\ [v/x]\ s'',\ t = t'\ \ !v\ t'', \\ & t' \text{ has no publication and } s' \text{ has no substitution for } x \\ \{\} & \text{otherwise .} \end{cases}$$

We lift the definition to $(S\ \textbf{where}\ x :\in T)$, where S and T are sets of traces.

$$(S\ \textbf{where}\ x :\in T) = \{p \mid p \in (s\ \textbf{where}\ x :\in t),\ s \in S,\ t \in T\}$$

Theorem 3 $\langle f\ \textbf{where}\ x :\in g\rangle = \langle f\rangle\ \textbf{where}\ x :\in \langle g\rangle$

Proof Sketch: The complete proof is in [7]. The proof is in two parts showing that one side is a subset of the other. If g never publishes, rule (ASYM1V) is never used; therefore, the operational behavior of $(f\ \textbf{where}\ x :\in g)$ is analogous that of $f \mid g$ because (ASYM1N) and (ASYM2) are the counterparts of (SYM1) and (SYM2), respectively. Then, any trace of $(f\ \textbf{where}\ x :\in g)$ is from $(s \mid t)$, where s and t are traces of f and g. If g has a trace $t'\ \ !v\ t''$ and f a trace $s'\ [v/x]\ s''$, then $(s' \mid t')s''$ is a trace of $(f\ \textbf{where}\ x :\in g)$, using the above argument and the meaning of substitution.
Note: Any substitution event $[v/x]$ in t is unrelated to x in $(s\ \textbf{where}\ x :\in t)$.

6 Monotonicity and Continuity of Combinators

The results of the last section show that the set of traces of any expression can be obtained from the traces of its constituent subexpressions. This motivates the following definition of *congruence* among expressions: two expressions are congruent, $\cong$, if their trace sets are equal. Similarly, define a partial order, $\sqsubseteq$, over expressions.

$f \cong g$ if $\langle f\rangle = \langle g\rangle$, and $f \sqsubseteq g$ if $\langle f\rangle \subseteq \langle g\rangle$

Each combinator preserves substitution of congruent subexpressions. That is, given $f \cong g$, we claim

1. $f \mid h \cong g \mid h$, and $h \mid f \cong h \mid g$
2. $f >x> h \cong g >x> h$, and $h >x> f \cong h >x> g$
3. f **where** $x :\in h \cong g$ **where** $x :\in h$, and
   h **where** $x :\in f \cong h$ **where** $x :\in g$

We prove the results by showing that each combinator is monotonic in both its arguments. That is, given $f \sqsubseteq g$, we prove the claims (1, 2, 3) with $\sqsubseteq$ replacing $\cong$. Then switching the roles of f and g, we get the congruences. We also prove continuity of the combinators. Underlying most of the proofs is the following lemma.

Lemma 4 Let each of $S_0, S_1, \cdots$ and T be a set of traces. Then,

1. $(\cup i :: S_i * T) = (\cup i :: S_i) * T$, where $*$ is any Orc combinator.
2. $(\cup i :: T * S_i) = T * (\cup i :: S_i)$, where $*$ is any combinator other than $>x>$.

Proof: From the definition of $*$ over trace sets, for arbitrary R and T,

$R * T = (\cup r : r \in R : r * T)$, where $*$ is any Orc combinator
$R * T = (\cup t : t \in T : R * t)$, where $*$ is any combinator other than $>x>$.

These follow from the lifting in the definition of the combinators over sets.

6.1 Monotonicity of Orc combinators

Each Orc combinator is monotonic in its left argument, e.g. $f \sqsubseteq g$ implies $f \mid h \sqsubseteq g \mid h$. This follows from Lemma 4, part(1); see [7] for details.

Monotonicity in the right argument for combinators other than $>x>$, (e.g. $f \sqsubseteq g$ implies $(h\ \textbf{where}\ x :\in f) \sqsubseteq (h\ \textbf{where}\ x :\in g)$), follows from Lemma 4, part(2). We give a proof that $f \sqsubseteq g$ implies $(h >x> f) \sqsubseteq (h >x> g)$ in [7].

6.2 Continuity of Orc combinators

Characterization of Least Upper Bound Let f denote a sequence of expressions $f_0, f_1, \cdots$, where $f_i \sqsubseteq f_{i+1}$, for all i. Expression F is an upper bound of f if for all i, $f_i \sqsubseteq F$, and F is the least upper bound of f if for any upper bound G of f, $F \sqsubseteq G$. Henceforth, we write $(\sqcup f)$ for the least upper bound of f. The proof of the following theorem is standard, and is given in [7].

Theorem 5 $\langle \sqcup f\rangle = (\cup i :: \langle f_i\rangle)$.

Continuity over left argument Let f be a sequence of expressions, and $h_i = f_i * g$, for all i, where $*$ is any Orc combinator. Then

$$\sqcup h \;\cong\; (\sqcup f) * g, \text{ i.e., } \langle \sqcup h \rangle = \langle (\sqcup f) * g \rangle.$$

The proof follows directly from Lemma 4, part(1).

Continuity over right argument Given a sequence g, and $h_i = f * g_i$, for all i, where $*$ is any Orc combinator, we show $\sqcup h \;\cong\; f * (\sqcup g)$. The proof follows directly from Lemma 4, part(2), where $*$ is any combinator other than $>x>$. For $>x>$, we prove the result in [7].

6.3 Least fixed point for recursive definitions

We have shown that $\sqsubseteq$ is a complete partial order over expressions. Next, we show that $\mathbf{0}$ is the bottom element. Any substitution $[v/x]$ is applicable to any expression because $f \overset{[v/x]}{\rightarrow} [v/x].f$. Hence, any sequence of substitutions D is a trace of any f (by applying induction on the length of D). Since $\mathbf{0}$ has no other transition,

$$\langle \mathbf{0} \rangle = \{D \mid D \text{ is a finite sequence of substitutions}\}$$

Therefore, for any f, $\langle \mathbf{0} \rangle \subseteq \langle f \rangle$, i.e., $\mathbf{0} \sqsubseteq f$.

Monotonicity and continuity of Orc combinators allow us to treat a recursively defined expression as the least upper bound of a chain of approximations. As an example, consider *Metronome* (described in Section 2.4) which we repeat below in an abbreviated form.

$$M \triangleq S \mid R \gg M$$

Then M is the least upper bound of the chain $M_0 \sqsubseteq M_1 \sqsubseteq \cdots$, where

$$M_0 = \mathbf{0}$$
$$M_{i+1} = S \mid R \gg M_i, \text{ for all } i, \text{ where } i \geq 0$$

6.4 A proof using congruence

Theorem 6 $f >x> let(x) \;\cong\; f$

Proof Sketch: The complete proof is in [7]. The proof is by structural induction on f. For the base cases (i.e. when f is any of $let(p)$, $?k$, $M(p)$, or $M(x)$), the result is proved by enumerating the traces (all maximal traces of f are of the form $r\ \ !v$, where r has no publications). For the inductive case, we consider three possible forms of f which are: $g \mid h$, $g >y> h$ and g **where** $y :\in h$. We apply certain laws from Section 4.1. From law (6), $(g \mid h) >x> let(x)$ is $g >x> let(x) \mid h >x> let(x)$, and inductively, this is $g \mid h$. From law (4), $(g >y> h) >x> let(x)$ is $g >y> (h >x> let(x))$, which is $g >y> h$, using induction on $h >x> let(x)$. From law (8), $(g$ **where** $y :\in h) >x> let(x)$ is $(g >x> let(x))$ **where** $y :\in h$, which is g **where** $y :\in h$, using induction on $g >x> let(x)$.

7 Related Work

There are two primary bodies of work on developing models for task orchestration. On the one hand, commercial workflow and orchestration languages have been the subject of formal study. On the other hand, traditional process algebra theory is being adapted to fit orchestration problems.

Petri nets have been proposed as a semantic model for workflow [8]. To compare commercial systems, Van der Aalst has proposed a set of workflow patterns [1]. These workflow patterns have been also been implemented in Orc [3] and the π-calculus [9]. Others have identified difficult patterns, like time-out [10], which have solutions in Orc.

A new Petri net language, YAWL, has been defined to express the patterns more directly [11]. YAWL's mechanism for multiple instantiation is analogous to Orc's sequential composition, but provides built-in synchronization. The node grouping and cancellation constructs are similar to Orc's **where** operator. Rather than build specific workflow patterns into the language, Orc provides just a few fundamental primitives with a mechanism to define new operators for user-defined composition patterns.

Process calculi, including CSP [12], CCS [6] and π-calculus [13], provide fundamental models of concurrency, with a focus on symmetric communication between threads. Orc has a structured approach to concurrency, and has an asymmetric relationship with its environment. Orc also supports a general sequential composition of expressions, $f \gg g$, and an explicit construct for process termination, which is synchronized with communication. Some variants of the π-calculus include a termination construct [14].

Simon Peyton-Jones suggested a relationship between Orc and the List monad as used in functional programming languages, including Haskell . The sequential composition operator, $>x>$, is analogous to the list bind >>=. The **where** operator resembles taking the first item from a lazy list. The standard list monad always produces values in a specific order, whereas the publication order in Orc is non-deterministic.

8 Conclusion

Task orchestration is a form of concurrent programming in which an agent invokes and coordinates the execution of passive, but potentially unreliable, services. Orchestration is well-suited to solving a range of concurrency problems, most notably workflow. Our practical experience in using Orc for orchestration has been very encouraging; we are able to code most concurrent programming paradigms succinctly. This paper shows that Orc has a simple trace semantics, and Orc combinators have many desirable properties such as monotonicity and continuity. The simplicity of the semantics may be a factor in simplicity of programming. We have addressed only the asynchronous aspects of Orc in this paper. We are now developing the full semantics which will combine asynchrony with time-based computations.

References

1. Aalst, W.M.P.V.D., Hofstede, A.H.M.T., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1) (2003) 5–51
2. Misra, J., Cook, W.R.: 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`.
3. Cook, W.R., Patwardhan, S., Misra, J.: Workflow patterns in Orc. In: Proc. of the International Conference on Coordination Models and Languages (COORDINATION). (2006) to appear.
4. Rosario, S., Benveniste, A., Haar, S., Jard, C.: SLA for web services orchestrations. Unpublished manuscript (2006)
5. Kozen, D.: On Kleene algebras and closed semirings. In: Proceedings, Math. Found. of Comput. Sci. Volume 452 of LNCS., Springer-Verlag (1990) 26–47
6. Milner, R.: Communication and Concurrency. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall (1989)
7. Kitchin, D., Cook, W.R., Misra, J.: Semantic properties of asynchronous Orc. Technical Report TR-06-32, University of Texas at Austin, Department of Computer Sciences (2006)
8. Eshuis, R., Dehnert, J.: Reactive Petri Nets for workflow modeling. In: International Conference on Applications and Theory of Petri Nets (ICATPN 2003). Volume 2679 of LNCS., Springer-Verlag (2003) 296–315
9. Puhlmann, F., Weske, M.: Using the π-calculus for formalizing workflow patterns. In: Proceedings of the 3rd International Conference on Business Process Management. Volume 3649 of LNCS. (2005)
10. van Glabbeek, R.: On specifying timeouts. In: Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond, Electronic Notes in Theoretical Computer Science (2005) to appear.
11. van der Aalst, W.M.P., Aldred, L., Dumas, M., ter Hofstede, A.H.M.: Design and implementation of the YAWL system. In Persson, A., Stirna, J., eds.: CAiSE. Volume 3084 of LNCS., Springer (2004) 142–159
12. Hoare, C.: Communicating Sequential Processes. Prentice Hall International (1984)
13. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999)
14. Riely, J., Hennessy, M.: Distributed processes and location failures. Theoretical Computer Science **266**(1–2) (2001) 693–735