Simulation using Orchestration
(Extended Abstract)

David Kitchin, Evan Powell, and Jayadev Misra

The University of Texas at Austin

Abstract. The real world is inherently concurrent and temporal. For simulating physical phenomena of the real world, one prefers frameworks which easily express concurrency and account for the passage of time. We propose *Orc*, a structured concurrent calculus, as a framework for writing simulations. Orc provides constructs to orchestrate the concurrent invocation of services while managing time-outs, priorities, and failures of services or communication. Orc's treatment of time is of particular interest in simulation. We propose an abstract notion of time and show its utility in coding simulations. We also show how Orc's structure allows us to compute statistics from a simulation.

1 Introduction

Orc[3, 4] 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 the combinators of the language. Naturally, Orc combinators support concurrency: parallel subcomputations, spawning of computations and blocking or termination of subcomputations. Orc has a number of algebraic properties which make it amenable to formal analysis.

Physical phenomena in the real world are inherently concurrent and temporal. Simulations of physical phenomena typically involve describing concurrent entities, their interactions and passage of real time. The structure of Orc makes such descriptions extremely modular. This paper presents a preliminary report of our experience with coding simulations in Orc.

In Section 2, we give a brief overview of Orc, followed by example Orc programs in Section 3. Portions of Sections 2 and 3 have appeared previously in [3, 4]. Section 4 presents an abstraction of time, in which we treat both physical (Newtonian) time and logical time analogously. In Section 5, we describe the implementation of simulations in Orc using logical timers. Section 6 describes how to compute statistics from a simulation. Section 7 includes plans for future research.

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. 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. Practical Orc examples use a modest set of syntactic extensions, discussed in Section 2.5. Orc examples appear in Section 3.

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.

We list a few sites in Figure 1 that are fundamental to effective programming in Orc (in the figure, a *signal* represents a unit value and has no additional information).

Site *if* is used for conditional evaluation. Site *Rtimer* is used to introduce delays and impose time-outs, and is essential for time-based computations. Site *Signal*() is a special case of *if*.

$if(b)$:	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.
Signal():	Returns a signal immediately. Same as $if(true)$.
0:	Blocks forever. Same as $if(false)$.

Fig. 1. Fundamental Sites

2.2 Combinators

There are three combinators in Orc for combining expressions. Given expressions f and g, they are: 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 then 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 with a value, x is bound to that value and $N(x)$ is then called. Contrast the following two 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 $E(\bar{x}) \triangleq f$ 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.

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

2.4 Formal Syntax

The formal syntax of Orc is given in Figure 2.[1] 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.

The syntax also allows actual parameters (variables x and values m) to appear alone as primitive expressions. The primitive expression m simply publishes m. The primitive expression x waits for the variable x to become bound, and then publishes the value bound to x.[2]

$$\begin{array}{rcll} f, g, h & \in & \textit{Expression} & ::= M(\bar{p}) \mid E(\bar{p}) \mid p \mid f >x> g \mid f \mid g \mid f <x< g \\ p & \in & \textit{Actual} & ::= x \mid m \\ & & \textit{Definition} & ::= E(\bar{x}) \triangleq f \end{array}$$

Fig. 2. Syntax of Orc

Notation The combinators are listed Figure 2 in decreasing order of precedence, so $f <x< g \mid h$ means $f <x< (g \mid h)$, and $f >x> g \mid h$ means $(f >x> g) \mid h$. Expression $f >x> g >y> h$ means $f >x> (g >y> h)$, i.e., $>x>$ is right-associative, and $f <x< g <y< h$ means $(f <x< g) <y< h$, i.e., $<x<$ is left-associative.

2.5 Syntax extensions

In practice, Orc programs often incorporate syntactic sugar, to simplify and condense expressions. These constructs do not fundamentally extend the calculus; they are simply more compact representations.

[1] Previous presentations of Orc have used the notation f **where** $x :\in g$ instead of $f <x< g$.

[2] Previous presentations of Orc used $let(x)$ to publish the value of x.

Tuples In additional to the fundamental sites shown earlier, it is also helpful to have site support for constructing and examining data structures. We allow the syntax $(\bar{p})$, as a shorthand for $\mathit{tuple}(\bar{p})$, where *tuple* is a site which creates a single tuple value out of its argument values and publishes it. In order to examine these tuples, we extend the syntax of the combinators $>x>$ and $<x<$ with *pattern matching*; instead of binding a value to a variable, a combinator may bind a tuple of values to a tuple of variables.

For example, the following expression publishes the values 6 and 7:

$$((3,6) \mid (4,7)) >(x,y)> y$$

Dot notation In some cases, especially when writing code in an object-oriented style, it is helpful to have a special notation for calls. We write $M.\mathit{name}$ as syntactic sugar for $M(\mathbf{name})$, where **name** is a message, and M maps messages to values. We use this notation to express both field accesses, written as $x.\mathit{field}$, and method calls, written as $x.\mathit{method}(\bar{p})$, which is a shorthand for $x.\mathit{method} >m> m(\bar{p})$.

Arithmetic and Logical Expressions Orc does not include any operators for data manipulation; so, 3+4 is an illegal expression in Orc. We get the same effect by calling a predefined site $\mathit{Sum}(3,4)$. To simplify coding, we write arithmetic and logical expressions in the standard way, like $3+4$, which are compiled into appropriate site calls.

3 Examples

We give a number of small examples in this section to familiarize the reader with the Orc style of programming. Most of these examples have appeared earlier, in [3] and [4].

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

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

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).

$$\begin{array}{l} z \ <z< \ (\ \ M() >x> (x, \mathit{true}) \\ \qquad\qquad \mid \ \mathit{Rtimer}(t) >x> (x, \mathit{false})\ ) \end{array}$$

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.

$$\begin{array}{lcll} forkjoin() & \triangleq & (x,y) & <x< M() \\ & & & <y< N() \end{array}$$

Synchronization There is no special machinery for synchronization in Orc; the $<x<$ combinator 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* (as described above), and start $f \mid g$ after *forkjoin* publishes.

$$forkjoin() \gg (f \mid g)$$

Delay 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 $Rtimer(1)$ and N.

$$Delay() \triangleq (Rtimer(1) \gg y) <y< N()$$

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 above,

$$x <x< (M() \mid Delay())$$

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 $c.get$ and $e.put$ are the methods to read from c and write to e. Below, $P(c,e)$ repeatedly reads from c and writes to e, and $Net(c,d,e)$ is a network of two such processes which share the output channel.

$$\begin{array}{rcll} P(c,e) & \triangleq & c.get() & >x> \; Compute(x) \\ & & & >y> \; e.put(y) \\ & & & \gg \quad P(c,e) \\ Net(c,d,e) & \triangleq & P(c,e) \mid P(d,e) & \end{array}$$

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 $or(x, y)$ returns $x \vee y$.

$$\begin{array}{lll} z & <z< & if(x) \gg true \mid if(y) \gg true \mid or(x,y) \\ & <x< & M() \\ & <y< & N() \end{array}$$

4 Timers

The site *Rtimer* is a powerful tool for orchestration. It is used mainly for orchestrating events that happen in real time, including interruptions (time-outs). However, the actual value of real time is never used in a computation. In this section, we consider a small combinatorial problem —computing the shortest path between two designated nodes in a weighted directed graph— for which we present a simple algorithm based on actual time values. Next, we introduce a more abstract version of timer, which is *logical* (or *virtual*) that mimics the (physical) real-time timer. We show that the shortest path problem can be solved using logical timers.

4.1 Shortest path algorithm using real time

Given is a directed graph each edge of which has a non-negative weight denoting the distance between the two nodes. There are two special nodes, designated *source* and *sink*. It is required to find a shortest path from the source to sink, i.e., one with the least total distance. Henceforth, we simply calculate the length of the shortest path; the actual shortest path can be computed by an easy extension.

The traditional algorithm for solving this problem, due to Dijkstra [1], involves inherently sequential computation. Consider, instead, the following real-time, concurrent algorithm. From the source node, transmit a ray of light to each of its neighbors. Rays propagate along each edge at constant speed in real time; the weight of each edge is the time taken by the ray to traverse the edge. When a node receives its first ray, it transmits a ray to each of its own neighbors.[3] Subsequent rays received by that node are ignored. The length of the shortest path is the total elapsed time from the start of the computation to the point where the sink node receives its first ray.

We code three different versions of this algorithm, with varying levels of refinement. First, we abstract the structure of the graph using expression *succ*:

$succ(u)$: Publish all pairs (v, d) where (u, v) is an edge with weight d.

The graph structure is completely characterized by the identities of the source and the sink, and expression *succ*.

[3] Assume that the amount of time to receive and rebroadcast a ray is inconsequential.

First Solution We must note when a node first receives a ray, and be sure to ignore subsequent rays. To implement this behavior, we associate a "write-once" variable with each node in the graph, and use two sites to manipulate these variables: For every node u in the graph, $write(u,t)$ writes t into u. Writing is once-only for each u; all subsequent writes block. And, $read(u)$ blocks until u is written; it never blocks subsequently, and returns the written value.

In the following algorithm, the first time $eval1(u,t)$ is called for any u, (1) the relative time in the evaluation is t, and (2) t is the length of the shortest path to u from the source. Note that $eval1$ does not publish.

$$eval1(u,t) \triangleq write(u,t) \gg Succ(u) >(v,d)> Rtimer(d) \gg eval1(v,t+d)$$

$$eval1(source,0) \mid read(sink)$$

Here, we write the value for the source at time 0. For any other node v, whose predecessor along the shortest path (from the source) is u, we write the value d time units after writing the value for u, where d is the weight of edge (u,v). And, we read the value written for the sink as soon as possible. We have assumed that executions of $Succ$, $read$ and $write$ do not consume any real time.

Second Solution The previous solution does not quite implement the real-time algorithm described for the problem. In particular, the path lengths are explicitly passed as parameter values; a node does not consult the elapsed time to record the length of the shortest path to it.

To this end, we associate real-time timers with computations. In the current implementation of Orc, calling $RealTimer$ site generates a new real-time timer and initializes its value to 0. Every generated timer runs in real time. Therefore, $Rtimer$ measures the progress of every real-time timer. To evaluate f with timer rt, write $RealTimer >rt> f$. Also, for each such timer rt, there is a site $rt.C$ that returns the current time of rt. The current time of rt is 0 when rt is created.

The following version of the shortest path algorithm is a more faithful rendering of the initial description. We have replaced $eval1(u,t)$ with $eval2(u,rt)$, where t can be computed from timer rt.

$$\begin{aligned} eval2(u,rt) \triangleq\ & rt.C() >t> write(u,t) \gg \\ & Succ(u) >(v,d)> Rtimer(d) \gg eval2(v,rt) \end{aligned}$$

$$RealTimer >rt> (eval2(source,rt) \mid read(sink))$$

Third Solution The previous solution records a time value for each node, whereas our interest is only in the shortest path to the sink. Therefore, we may simplify the recording for the nodes. Instead of $write(u,t)$, we use $mark(u)$ which merely notes that a node has been reached by a ray of light. Similarly, instead of $read(u)$, we employ $scan(u)$ which responds with a signal if u has been marked. The length of the shortest path is the value of $rt.C()$ when the sink is marked.

$$eval3(u,rt) \triangleq mark(u) \gg Succ(u) >(v,d)> Rtimer(d) \gg eval3(v,rt)$$

$$RealTimer >rt> (eval3(source,rt) \mid scan(sink) \gg rt.C())$$

4.2 Logical or Virtual timers

Each of the shortest path algorithms given in the previous section waits for real time intervals. The running time of each algorithm is proportional to the actual length of the shortest path. This is quite inefficient. Additionally, We have assumed that executions of *Succ*, *read* and *write* do not consume any real time, which is unrealistic. To overcome these problem, we explore the use of logical (virtual) timers to replace real-time timers.

There are three essential properties of real-time timers that we have used in the previous section. Let rt be a real-time timer, and, as before, $rt.C()$ returns the current value of this timer.

1. (Monotonicity) The values returned by successive calls to $rt.C()$ are non-decreasing.
2. (Relativity) Using a notation similar to Hoare-triples, where $rt.C()$ denotes the value returned by a call to $rt.C()$,

$$\{rt.C() = n\}\ Rtimer(t)\ \{rt.C() = n + t\}$$

3. (Weak Progress) Some call to $Rtimer(.)$ responds eventually.

Monotonicity guarantees that $s \leq t$ in $rt.C()\ {>}s{>}\ \cdots rt.C()\ {>}t{>}\ \cdots$. Relativity says that if $Rtimer(t)$ is called when a timer value is n, the response to the call is received at time $n + t$. This property establishes the essential relationship between $rt.C$ and $Rtimer$. The progress property is a weak one, merely postulating the passage of time. Typically, we need a Strong Progress property: *every* call to $Rtimer(t)$ responds eventually. However, it can not be met in arbitrary Orc programs where infinite number of events may take place within bounded time, as in the following examples.

$$\begin{array}{rcl} Met() & \underline{\Delta} & Signal \mid Rtimer(0) \gg Met() \\ M() & \underline{\Delta} & N() \mid M() \end{array}$$

It is the obligation of the programmer to ensure that only a finite number of events occur during any finite time interval. A sufficient condition is that every recursive call is preceded by a call $Rtimer(t)$, where t is a positive integer. Then we can guarantee the Strong Progress property.

A logical (or virtual) timer is generated by a call to site $VirtTimer()$. There are two site calls associated with a logical timer lt: $lt.C()$ and $lt.R(t)$. These calls are analogous to $rt.C()$ and $Rtimer(t)$ for real-time timers. Further, logical timers obey the requirements of Monotonicity, Relativity and Weak Progress, as for the real-time timers. They also obey the Strong Progress property under analogous assumptions. We show in Section 4.3 how logical timers may be implemented.

There is one key difference between real-time and virtual-time timers. For site M other than a timer site, no logical time is consumed between calling the site and receiving its response, whereas real time may be consumed. Conversely, no real time is consumed in any interval where logical time is consumed.

We can rewrite the solutions to the shortest path problem using logical timers. Below, we do so for the third solution in Section 4.1, using $VirtTimer()$ to generate a virtual timer.

$$eval4(u, lt) \triangleq mark(u) \gg Succ(u) >(v, d)> lt.R(d) \gg eval4(v, lt)$$

$$VirtTimer() >lt> (eval4(source, lt) \mid scan(sink) \gg lt.C())$$

Observe that *mark*, *scan* and *Succ* may consume real time, though they do not consume any logical time. Further, the actual computation time is now decoupled from the length of the shortest path. Dijkstra's shortest path algorithm [1] is a sequential simulation of this algorithm that includes an efficient implementation of the logical timer.

4.3 Implementing logical timer

Let lt be a logical timer. Associate a value n with lt. Initially (when lt is created) $n = 0$. A call to $lt.C()$ responds with n. Call $lt.R(t)$ is assigned a *rank* $n + t$ and queued (in a priority queue). Eventually, the timer responds with a signal for the item of the lowest rank r in the queue, if an item exists, and removes it from the queue. Simultaneously, it sets n to r.

It is easy to see that Monotonicity holds, that n never decreases: we have the invariant $n \leq s$, for any rank s in the queue, and that the latest response to $lt.C()$ is n. Similarly, Relativity is also easy to see. The requirement of Weak Progress is met by eventually removing an item from the queue. Further, if only a finite number of events occur in any bounded logical time interval, the Strong Progress property is also met.

Note that the specification of a timer only ensures that the timer's responses are properly ordered with respect to each other. The relationship between a timer's responses and the behavior of other sites (or timers) is unspecified. This gives us a great deal of flexibility in implementing timers.

4.4 Stopwatch

A *stopwatch* is a site that is aligned with some timer, real or virtual. We will see some of its uses in simulation in Section 5.

A stopwatch is in one of two states, *running* or *stopped*, at any moment. It supports 4 methods: (1) *reset*: is applicable when the stopwatch is stopped, and then its value is set to 0; (2) *read*: returns the current value of the stopwatch; (3) *start*: changes the state from stopped to running; and (4) *stop*: changes the state from running to stopped.

A stopwatch can be implemented by having the variables *running* (boolean) and m, n (integer). The current state is given by *running*. If $\neg running$ holds (i.e., the state is stopped), then m is the value of the stopwatch and n's value is irrelevant. If *running* holds, then m is the value of the stopwatch when it was last started, and n is the value of lt, the timer with which the stopwatch

is aligned, when the stopwatch was last started. Initially, *running* is *false* and both m and n are zero. The methods are implemented (in imperative-style) as follows.

reset: $m := 0$
read: **if** *running* **then** return$(m + lt.C() - n)$ **else** return(m)
start: $running := true$; $n := lt.C()$
stop: $running := false$; $m := m + lt.C() - n$

Note: Only the *read* method responds to its caller. The other methods do not respond, though they update the internal state of the stopwatch.

5 Simulation

The work reported in this section is at a preliminary stage.

A *simulation* is an abstraction of real-world processes. The goal of simulation is to observe the behaviors of the abstract processes, and compute statistics. A faithful simulation can predict the behavior of the real-world processes being simulated. A simulation language supports descriptions of the real-world processes, their interactions and the passage of real time. We contend that Orc is an effective tool for writing simulations. We can describe the individual processes as expressions in Orc. As we have demonstrated with the shortest path example in Section 4, replacing real-time timer with a logical timer can efficiently simulate the passage of time while maintaining the expected causal order among events.

Orc also simplifies data collection and statistics computation of the simulated processes because of its structured approach to concurrency. Since the lexical structure of the program reflects the dynamic structure of the computation, it is easy to identify points in the program at which to add observations and measurements. In an unstructured model of concurrency, this can be a more challenging task.

We show two small examples of simulation in Orc in this section. The examples, though small, are typical of realistic simulations. We consider data collection in the following section.

5.1 Example: Serving Customers in a Bank

Consider a bank that has two tellers to serve customers. A stream of customers arrive at the bank according to some arrival distribution. Each customer joins a queue on entering the bank. A teller asks the next customer to step forward whenever she is free. The service time for a customer is determined by the type of transaction. It is required to determine the average wait time for a customer, the queue length distribution, and the percentage of time that a teller is idle. In this section, we merely represent the system using Orc; computation of statistics is covered in the following section.

We represent the bank as consisting of three concurrent activities, customers and two tellers. We define each of these activities by an expression. Customers are generated as a stream by expression *Source* according to some given distribution. This expression also specifies the service time of each customer. We do not code *Source* though a complete simulation would have to include it.

The goal expression, given at the top of Figure 3, starts a logical timer lt, runs expression *Bank* for *simtime* logical time units (using the time-out paradigm), and then publishes the statistics by calling $Stats()$. Observe that expression $Bank()$ does not publish, which is ensured by sequential composition with $\mathbf{0}$, permitting us to use the time-out paradigm.

$$
\begin{array}{rcl}
 & & VirtTimer() >lt> \\
 & & (z <z< Bank(lt) \mid lt.R(simtime)) \gg \\
 & & Stats() \\
Bank(lt) & \triangleq & (Customers() \mid Teller(lt) \mid Teller(lt)) \gg \mathbf{0} \\
Customers() & \triangleq & Source() >c> enter(c) \\
Teller(lt) & \triangleq & next() >c> \\
 & & lt.R(c.ServTime) \gg \\
 & & Teller(lt) \\
enter(c) & \triangleq & q.put(c) \\
next() & \triangleq & q.get()
\end{array}
$$

Fig. 3. Bank simulation

The Orc definitions have mostly described a physical system; therefore it is extremely succinct. The description is modular, which allows for experimentation with a variety of policies (e.g., assigning one teller to handle short jobs, for instance) and different mixes of system parameters (e.g., hiring more tellers). Further, there is no explicit mention of simulation in the definitions, only in the goal expression. Advancing of the logical timer will be automatically handled by the implementation.

5.2 Example: Serving Customers in a Fast food restaurant

The next example, serving customers in a fast food restaurant, is similar to that of the bank, though there are key differences. As in the bank example, we have a steady stream of customers entering a queue, and we have a single cashier in place of tellers. Rather than servicing customers' orders directly, the cashier processes the orders and puts them in another queue to be handled by one of the two cooking stations. Cooking stations prepare the main entree, side dish and the drink parts of an order in parallel, where each part takes some amount of time to complete. An order is complete only after all its parts have been completed. Unlike the bank example where each customer carried its service time with it, we let the restaurant decide the service time for each customer c, by calling $ringupTime(c)$ to determine the cashier's time, and $prepTime(c.drink)$ for the

time required to prepare the drink order for c (and, similarly, for the other parts of the order).

Figure 4 describes the simulation of the restaurant for *simtime* logical time units. Note that *Cook* uses the fork-join strategy discussed in Section 3 (we have abbreviated a cooking station by *Cook*). Both q and $orders$ are FIFO channels which we use for our queues. Analogous to $enter(c)$ and $next()$, we could have entered and removed orders indirectly in queue $orders$ rather than directly as we do in Figure 4.

$$
\begin{array}{rcl}
 & & VirtTimer() >lt> \\
 & & (z <z< Restaurant(lt) \mid lt.R(simtime)) \gg \\
 & & Stats() \\
Restaurant(lt) & \triangleq & (Customers() \mid Cashier(lt) \mid Cook(lt) \mid Cook(lt)) \gg \mathbf{0} \\
Customers() & \triangleq & Source() >c> enter(c) \\
Cashier(lt) & \triangleq & next() >c> \\
 & & lt.R(c.ringupTime) \gg \\
 & & orders.put(c.order) \gg \\
 & & Cashier(lt) \\
Cook(lt) & \triangleq & orders.get() >order> \\
 & & (\\
 & & \quad (e, s, d) \\
 & & \quad\quad <e< prepTime(order.entree) >t> lt.R(t) \\
 & & \quad\quad <s< prepTime(order.side) \quad >t> lt.R(t) \\
 & & \quad\quad <d< prepTime(order.drink) >t> lt.R(t) \\
 & &) \gg \\
 & & Cook(lt) \\
enter(c) & \triangleq & q.put(c) \\
next() & \triangleq & q.get()
\end{array}
$$

Fig. 4. Fast food restaurant simulation

6 Measurement

The typical purpose of simulation is to measure the behaviors exhibited by the simulated processes. These measurements are especially useful when they incorporate information about the passage of time in the simulation, such as the total amount of time that a participant remained idle or the average delay experienced by some process waiting on another process.

The current time associated with logical timer lt is $lt.C$. We use this value to report the times at which certain events occur in the simulation. We also use differences between observed times to determine the duration of some activity.

Consider a fragment of the bank example, where we are adding customers to the queue and later removing them:

$$
\begin{array}{rcl}
enter(c) & \triangleq & q.put(c) \\
next() & \triangleq & q.get()
\end{array}
$$

We augment this part of the simulation with measurements to determine the amount of time each customer spends waiting in line. We report the waiting time with the site *reportWait*.

$$
\begin{array}{rcl}
enter(c) & \triangleq & lt.C >s> q.put(c, s) \\
next() & \triangleq & q.get() >(c, t)> \\
 & & lt.C >s> \\
 & & reportWait(s - t) \gg \\
 & & c
\end{array}
$$

Histogram We can also compute histograms or queue length distribution, as follows. Let t_i, where $0 \leq i < N$, be the duration during simulation for which the length of q has been i. We create $N + 1$ stopwatches, $sw[0..N]$, at the beginning of simulation. The final value of $sw[i]$, $0 \leq i < N$, is t_i. And, $sw[N]$ is the duration for which the queue length is at least N.

Now, modify $enter(c)$ and $next()$ to ensure that whenever the queue length is i, $0 \leq i < N$, $sw[i]$ is running and all other stopwatches are stopped (similarly for $sw[N]$). Therefore, initially, only $sw[0]$ is running. Whenever a new item is added to a queue of length i, $0 \leq i < N$, we stop $sw[i]$ and start $sw[i + 1]$; for $i = N$, nothing needs to be done. Similarly, after removing an item if the queue length is i, $0 \leq i < N$, we start $sw[i]$ and stop $sw[i + 1]$.

The modifications to $enter(c)$ and $next()$ are shown below. Assume $q.length$ returns the current length of q. Note that the code fragment
$q.length >i> if(i < N) \gg (sw[i].stop \mid sw[i + 1].start)$
does not publish.

$$
\begin{array}{rcl}
enter(c) & \triangleq & lt.C >s> q.put(c, s) \gg \\
 & & q.length >i> if(i < N) \gg (sw[i].stop \mid sw[i + 1].start) \\
next() & \triangleq & q.get() >(c, s)> \\
 & & (\ lt.C >t> reportWait(s - t) \gg c \\
 & & \mid q.length >i> if(i < N) \gg (sw[i].start \mid sw[i + 1].stop) \\
 & &)
\end{array}
$$

7 Summary and Conclusions

This paper reports some preliminary work on coding simulations in Orc. Orc supports descriptions of concurrent activities and real time, which make it possible to describe many physical systems. We have introduced logical timers in this paper to facilitate computations that do not need to synchronize with the wall clock. We have described some of the properties of logical timers and shown their use in solving a combinatorial problem (shortest path) as well as in coding simulations.

Orc cannot succinctly express certain simulations because it does not have the fundamental notion of *guarded choice*, as found in the π-calculus [2] and other concurrent calculi. For example, a *Teller* that watches two queues and takes a customer whenever either queue becomes non-empty is difficult to code without such a choice combinator. The addition of guarded choice to Orc is a topic of ongoing research.

References

1. E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:83–89, 1959.
2. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
3. J. Misra and W. 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`.
4. I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. A timed semantics of orc. *Theoretical Computer Science*, 2008. To appear.