Reduction Semantics and Formal Analysis of Orc Programs

Musab AlTurki [1]   José Meseguer [2]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Abstract

Orc is a language for orchestration of web services developed by J. Misra that offers simple, yet powerful and elegant, constructs to program sophisticated web orchestration applications. The formal semantics of Orc poses interesting challenges, because of its real-time nature and the different priorities of external and internal actions. In this paper, building upon our previous SOS semantics of Orc in rewriting logic, we present a much more efficient *reduction semantics* of Orc, which is provably equivalent to the SOS semantics thanks to a strong bisimulation. We view this reduction semantics as a key intermediate stage towards a future, provably correct distributed implementation of Orc, and show how it can naturally be extended to a distributed actor-like semantics. We show experiments demonstrating the much better performance of the reduction semantics when compared to the SOS semantics. Using the Maude rewriting logic language, we also illustrate how the reduction semantics can be used to endow Orc with useful formal analysis capabilities, including an LTL model checker. We illustrate these formal analysis features by means of an online auction system, which is modeled as a distributed system of actors that perform Orc computations.

Keywords: Orc, rewriting logic, formal verification, real-time semantics, web services

1 Introduction

At present, the reliability of web-related software is poor, to say the least; and formal analysis is one of the most effective ways to increase the quality, reliability, and security of webware. For example, formal specification and model checking analysis of Internet Explorer has uncovered many, previously unknown, types of address-bar and status bar spoofing attacks [8]. There is however, a substantial gap between the level of the formal specifications readily amenable to analysis, and the low level implementations of webware in conventional languages. This gap can be narrowed by the use of model checkers for conventional languages such as Java or C, which may be a reasonably practical way, though hard to scale up, to verify legacy

[1] Email: alturki@cs.uiuc.edu
[2] Email: meseguer@cs.uiuc.edu

systems. But such a conventional approach to the design of webware is not, by any means, the best way to design and verify future web-based systems.

This work is part of a longer-term research effort to explore a new webware design and implementation approach based on two main ideas: (i) the systematic use of formal executable specifications in rewriting logic to precisely capture the intended semantics and to verify relevant properties; and (ii) the stepwise refinement of such specifications into a provably correct distributed implementation. It helps of course very much to begin with a type of webware that is mathematically elegant, simple, novel, and promising in its practical applications. We have focused on J. Misra's Orc language, a simple and elegant language to orchestrate complex web services [21,22,14]. In spite of its inherent simplicity, the formal semantics of Orc presents interesting challenges. These challenges center around two main aspects of the Orc semantics: (i) the inherent priority that internal actions should have over external communication events; and (ii) the real-time nature of the language. In particular, a standard SOS semantics is insufficient to capture the intended Orc semantics [22,1].

In a previous paper [1], we have used rewriting logic to capture the intended semantics of Orc at the highest level possible, presenting an SOS-like rewriting semantics that does justice to Orc's real-time features. In this work we take two steps towards the refinement of Orc specifications into a distributed implementation. Our first and most crucial step is the refinement of our original SOS-like Orc semantics into a much more concurrent *reduction semantics*, which takes full advantage of rewriting logic's concurrent semantics, fully exploits rewriting logic's crucial distinction between equations and rewrite rules. Our three main contributions for this first step are: (i) showing how the real-time synchronous semantics of Orc can be faithfully captured in the reduction semantics; (ii) establishing the semantic equivalence between the reduction semantics and the SOS-like semantics; and (iii) providing experimental evidence for the claim that the reduction semantics is much more efficient than the SOS-like semantics.

The second refinement step is a simple, yet important extension of the first. The Orc semantics as such focuses on the, possibly concurrent, evaluation of a single Orc expression, abstracting away its interactions with external sites as "black boxes" in an external environment. It is however very natural to view both Orc expressions and sites as *distributed objects*, which interact with each other through message passing. Therefore, in this second refinement step we encapsulate both Orc expressions and sites as distributed objects, essentially reusing the already given reduction semantics in the semantic specification of Orc expression objects. All this can be done easily and naturally by using rewriting logic's approach to distributed objects [17]. Although still a specification, this distributed object semantics brings the Orc refinement quite close to a future distributed implementation. Our work also shows how nontrivial formal analyses of relevant Orc applications can be carried out with good efficiency, even after these two steps of refinement. Specifically, we show how Maude's LTL model checker can be used to verify the requirements of an online Orc auction system realized as a distributed collection of Orc expression and site objects.

The paper is organized as follows. Section 2 discusses related work and presents preliminaries on Orc, rewriting logic, and Maude. The reduction semantics is then presented in Section 3. Its further refinement into a distributed object-based semantics and the verification of an online auction case study are covered in Section 4. Conclusions and future work are discussed in Section 5.

2   Related Work and Preliminaries

Several Orc semantics have already been given. A precise but informal operational semantics for Orc was given by Misra in [21]; we consider this as the standard against which the success of any formal operational semantics should be measured. A formal SOS *asynchronous* operational semantics has been given by Misra and Cook in [22]; but since this asynchronous semantics allows some undesirable behaviors, a refinement of the asynchronous semantics into a *synchronous* semantics, distinguishing between *internal* and *external* actions was also given in [22]. Different denotational semantics of Orc for reasoning about identities and algebraic laws about the language [13,14], and for formally analyzing dependencies in program execution [25] were also proposed. Moreover, encodings of Orc in Petri nets and the join calculus that reveal some of the subtleties of the semantics of the language were given in [5]. Most recently, Ian Wehrman et al. [27] proposed a relative-time operational semantics of Orc by extending the asynchronous SOS relation of [22] to timed events and time-shifted expressions.

Our work, along with some of the operational approaches cited above, has similarities with the various SOS semantics that have been given for different timed process calculi, such as ATP [23] and TLP [12], and real-time extensions to various process calculi, such as extensions of ACP [3,4], CCS [7], and CSP [26].

2.1   The Orc Programming Model

Orc is a theory of orchestration, proposed by J. Misra [21], to model the smooth integration of web services. The Orc model is fairly minimal, yet powerful enough to express a wide range of computations [22]. Orc is based on the abstract notion of sites and the composition of the services they provide. A *site* is a basic service that provides a computation of some kind. For instance, $CNN(d)$ and $BBC(d)$ are sites that return the news for the given date d, and $add(x, y)$ is one that returns the sum of its arguments. Sites are assumed to exist, and the computations they provide constitute the data processed by Orc expressions. A site, when called, produces at most one value. When a site responds to a call with a value v, the site is said to *publish* the value v. Moreover, site calls are *strict*, in the sense that a site call cannot be initiated before its parameters are bound to concrete values.

There are six fundamental sites that are available to any Orc program. These sites and the services they provide are shown in Figure 1, assuming t is a nonnegative integer, b is a boolean, and x and y are values of arbitrary types.

The extended Orc syntax we use here is given in Figure 2. An Orc *program* consists of an optional list of declarations followed by an Orc expression. A *dec-*

$let(x, y, \dots)$	returns a tuple consisting of its argument values.
$clock$	returns the current time as a non-negative integer.
$atimer(t)$	returns a signal at time t.
$rtimer(t)$	returns a signal after t time units.
$signal$	returns a signal immediately.
$if(b)$	returns a signal if b is true; otherwise it remains silent.

Fig. 1. Fundamental sites in Orc

D	$\in$	$d_1; \dots; d_n$ (A list of declarations)	Orc program	$::=$	$D \; ; \; f$	
E	$\in$	Expression Name	$d \in$ Declaration	$::=$	$E(Q) =_{def} f$	
x	$\in$	Variable	$f, g \in$ Expression	$::=$	$\mathbf{0} \mid M(P) \mid E(P)$	
M	$\in$	Site		$\mid$	$f \mid g$	
c	$\in$	Constant		$\mid$	$f > x > g$	
h	$\in$	Handle		$\mid$	g **where** $x :\in f$	
P	$\in$	$p_1, \dots, p_n$ (A list of actuals)		$\mid$	$! \, c \mid \, ! \, x \mid \, ? \, h$	
Q	$\in$	$q_1; \dots; q_n$ (A list of formals)	$p \in$ Actual Parameter	$::=$	$x \mid c \mid M$	
			$q \in$ Formal Parameter	$::=$	x	

Fig. 2. Extended syntax of Orc

laration consists of a name, a (possibly empty) list of formal parameters, and an expression representing its body. An *expression* can be either: (1) the silent site ($\mathbf{0}$), which is a site that never responds; (2) a site or an expression call having an optional list of actual parameters; (3) the publishing of a value or a variable; (4) a placeholder for an unfinished site call; or (5) the composition of two expressions by one of the three composition operators. Two expressions may be composed either sequentially with the sequencing operator $> \; x \; >$, or in parallel. Parallel composition comes in two flavors: (1) symmetric composition, using $\mid$, where multiple threads execute concurrently returning a (possibly empty) stream of values; and (2) asymmetric composition, using the **where** statement, in which the left expression executes concurrently with possibly many threads of the right expression, choosing the first result published by any one of them.

A formal description of the (untimed) asynchronous semantics of Orc was given as an SOS specification in [22], and is shown here in Figure 3. The reader is referred to [22], for a detailed discussion of the specification. Here, only a few subtleties are emphasized. We first note that symmetric parallel composition $f \mid g$ in Orc is similar to that of a process calculus in which both expressions f and g can evolve concurrently without any restriction. However, in asymmetric parallel composition g **where** $x :\in f$, once f publishes its first value (the Asym1V rule), the remaining computations of f are discarded and the published value is bound to x in g. Therefore, it is possible for some computations of g to be blocked waiting for a value for x. We also note that in sequential composition $f > x > g$, a new instance of g is created for every value published by f (the Seq1V rule), which generalizes the usual notion of sequential composition in sequential programming languages. For example, consider the following program, which can be found, among many

$$\frac{h \text{ fresh}}{M(c) \overset{M\langle c,h\rangle}{\hookrightarrow} ?h} \text{ (SiteCall)} \qquad \frac{f \overset{!c}{\hookrightarrow} f'}{f > x > g \overset{\tau}{\hookrightarrow} (f' > x > g) \mid g\{c/x\}} \text{ (Seq1V)}$$

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

$$!c \overset{!c}{\hookrightarrow} \mathbf{0} \qquad \text{ (Pub)}$$

$$\frac{E(Q) =_{def} f \in D}{E(P) \overset{\tau}{\hookrightarrow} f\{P/Q\}} \quad \text{ (Def)} \qquad \frac{f \overset{!c}{\hookrightarrow} f'}{g \text{ where } x :\in f \overset{\tau}{\hookrightarrow} g\{c/x\}} \text{ (Asym1V)}$$

$$\frac{f \overset{l}{\hookrightarrow} f'}{f \mid g \overset{l}{\hookrightarrow} f' \mid g} \quad \text{ (Sym1)} \qquad \frac{f \overset{l}{\hookrightarrow} f' \quad l \neq !c}{g \text{ where } x :\in f \overset{l}{\hookrightarrow} g \text{ where } x :\in f'} \text{ (Asym1N)}$$

$$\frac{g \overset{l}{\hookrightarrow} g'}{f \mid g \overset{l}{\hookrightarrow} f \mid g'} \quad \text{ (Sym2)} \qquad \frac{g \overset{l}{\hookrightarrow} g'}{g \text{ where } x :\in f \overset{l}{\hookrightarrow} g' \text{ where } x :\in f} \text{ (Asym2)}$$

Fig. 3. Asynchronous semantics of Orc

other examples, in [21,22].

$$DelayedN =_{def} (rtimer(1) > x > let(u)) \text{ where } u :\in N;$$

$$let(x) \text{ where } x :\in (M \mid DelayedN)$$

The program implements a prioritized site call. Site M is given priority over site N, in that a response from M, if received within one time unit, would be the value published by the expression. Otherwise, either value published by M or N is published.

2.2 Rewriting Logic and Maude

Rewriting logic [16] is a general semantic framework that unifies in a natural way a wide range of models of concurrency. In particular, it is well suited to both give formal semantic definitions of programming languages, including concurrent ones (see [15,20] and references there), and to model real-time systems [24]. Furthermore, with the availability of high-performance rewriting logic implementations, such as Maude [10], language specifications can both be executed and model checked.

A rewrite theory is a formal description of a concurrent system, including its static state structure and its dynamic behavior. In its most general form, a *rewrite theory* is a 4-tuple $\mathcal{R} = (\Sigma, E, R, \phi)$ with:

- (Σ, E) a membership equational logic (MEL) theory [18], with Σ a MEL signature having a set of kinds, a family of sets of operators, and a family of disjoint sets of sorts, and E a set of Σ-sentences, which are universally quantified Horn clauses with atoms that are equations $(t = t')$ and memberships $(t : s)$, with t, t' terms and s a sort,

- R a set of universally quantified labeled *conditional rewrite rules* of the form:

$$(\forall X) \; r : t \to t' \text{ if } \bigwedge_i p_i = q_i \wedge \bigwedge_j r_j : s_j \wedge \bigwedge_l w_l \to w'_l$$

where r is a label, $t, t', p_i, q_i, r_j, w_l$ and w'_l are terms, and s_j are sorts.

- $\phi : \Sigma \to \mathcal{P}(\mathbf{N})$ a function that assigns to each operator symbol f in Σ of arity $n > 0$ a set of positive integers $\phi(f) \subseteq \{1, \ldots, n\}$ representing *frozen* argument positions where rewrites are forbidden.

A rule in R gives a general pattern for a possible change or transition in the state of a concurrent system. Changes are deduced according to the set of inference rules of rewriting logic, which are described in detail in [6]. Using these inference rules, a rewrite theory $\mathcal{R}$ proves a statement of the form $(\forall X)\ t \to t'$, written as $\mathcal{R} \vdash (\forall X)\ t \to t'$, meaning that, in $\mathcal{R}$, the state term t can transition to the state term t' in a finite number of steps. A detailed discussion of rewriting logic as a unified model of concurrency and its inference system can be found in [16]. [6] gives a precise account of the most general form of rewrite theories and their models.

Maude is a high-performance implementation of rewriting logic and its underlying MEL sublogic, with syntax that is almost identical to the mathematical notation. A basic unit of specification in Maude can either be a *functional module*, corresponding to a MEL theory (Σ, E), or a *system module*, representing a rewrite theory (Σ, E, R, ϕ). Besides the ability to execute a system module's specification (using the `rewrite` command) and to systematically search its state space (using the breadth-first `search` command), Maude provides an LTL model checker to verify complex LTL safety and liveness properties about finite state systems. For a complete description of Maude and its features, the reader is referred to [9].

3 Reduction Semantics of Orc

In previous work [1], we have developed a formal semantics of Orc in rewriting logic based on the SOS specifications of Figure 3. The semantics was in some sense a direct translation of the SOS specifications into a rewrite theory $\mathcal{R}^{sos}_{Orc}$ using a semantics-preserving MSOS-to-rewriting logic transformation [19]. Therefore, the rewriting semantics given by $\mathcal{R}^{sos}_{Orc}$ is readily understandable and its correctness is an immediate result of the correctness of the transformation used. However, $\mathcal{R}^{sos}_{Orc}$ makes extensive use of conditional rewrite rules (corresponding to the rules in the SOS specifications) which cannot be converted into equations without destroying the correctness of the semantics. Moreover, most of these rewrite rules, besides being conditional, have rewrites in their conditions, which is typical of the SOS specification style. In practice, this means that their execution, which uses breadth-first search to satisfy the rewrite conditions, is quite expensive and inefficient. In addition, all the rules in $\mathcal{R}^{sos}_{Orc}$ are system-wide rules defined at the configuration (state) level, forcing an interleaving semantics and not exploiting rewriting logic's features to express concurrent computations.

In this section we develop a rewriting semantics specification of Orc that, unlike $\mathcal{R}^{sos}_{Orc}$, is not based on the structural operational semantic rules of Figure 3, but is instead based on the inherently distributed semantics of rewriting logic. This rewriting semantics is in the style of what is usually called *reduction semantics*, but has the added advantage of using both equations and rules, thus achieving a simpler,

more flexible semantics and a smaller state space, since only transitions caused by rules create new states in the state space. The proposed specification, which we will henceforth call $\mathcal{R}_{Orc}^{red}$, is still operational, in that it describes in detail how Orc programs are evaluated, and is, in fact, semantically equivalent to $\mathcal{R}_{Orc}^{sos}$, in the sense that, given any Orc program P, the state transition systems of the semantics of P given by $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ are strongly bisimilar. However, by minimizing the number of rewrite rules and reducing their complexity, we achieve a simpler and indeed superior semantic specification that can be executed and analyzed much more efficiently. Before discussing the rewriting semantics specifications, we briefly describe the semantics infrastructure required (see [2] for a detailed description).

3.1 Semantic Infrastructure

A state in the evolution of an Orc program is represented with an Orc configuration. A *configuration* is a pair $\langle f, r \rangle$ with f an Orc program and r a *record*, which is a set of fields (built with an associative-commutative set union operator $\_|\_$) representing the different semantic entities required to correctly capture the semantics of f, such as environments and stores. By abstracting such semantic entities with the notion of fields, we obtain generality and modularity in the specification of language semantics, as explained in [19]. In particular, there are five fields in an Orc configuration, which are briefly described below.

Messages: The messages field is of the form $(msg : \rho)$, where ρ is a pool (multiset) of messages in transit, modeling site calls generated by the expression component f of the configuration, and incoming site returns from the environment.

Context: The context field $(con : \sigma)$ is an environment mapping expression names to expression definitions. An expression definition in σ may refer to names in the environment allowing definitions of (mutually) recursive Orc programs.

Handle: The handle field $(hdl : h_n)$ maintains the next available handle name h_n, with n a natural number. By the SITECALL rule of Figure 3, fresh handle names need to be generated when site calls take place. Handle names serve as identifiers for pending site calls; a handle identifies which message in the message pool ρ belongs to a given unfinished site call in the Orc program f.

Trace: The trace field $(tr : t)$ records a list of events t representing the transitions undergone by the Orc program[3]. The four event types are site calls, site returns, publishing of values, and the unobservable event τ.

Clock: The clock field $(clk : c_m)$ maintains a discrete time global clock modeled using the domain of natural numbers (m is a natural number). We shall see later in Section 3.4 how the clock field is used to give the timed semantics of Orc.

Therefore, based on the description given above, the general form of an Orc configuration is $\langle f, msg : \rho \mid con : \sigma \mid tr : t \mid hdl : h_n \mid clk : c_m \rangle$.

---

[3] Unlike the SOS-based rewriting semantics of [1], the trace field is entirely optional for the semantics described in this paper. However, in order to preserve equivalence with the aforementioned semantics, we opt to keep this field in the specifications discussed in this section (see Section 3.5).

3.2   Rewrite Rules and Equations

We now specify the rewrite theory $\mathcal{R}_{Orc}^{red} = (\Sigma, E, R, \phi)$ using the above semantic infrastructure. $\mathcal{R}_{Orc}^{red}$ captures four actions an Orc configuration can take: (1) calling a site, (2) calling an expression, (3) publishing a value, and (4) returning a value from a site. In this section, we list and discuss all the rules R and some of the equations of E. A complete discussion of all the equations can be found in [2].

A common characteristic of the specifications of the actions mentioned above is the need to propagate information back and forth between a subterm of an Orc expression and the configuration it is contained within. This propagation of information is specified using auxiliary functions that are defined inductively on the structure of an expression. In the following paragraphs, we describe these actions and auxiliary functions in some detail.

Site Call. A site call is modeled by the following two rewrite rules.

$$\text{SITECALL} : \langle f, r \rangle \rightarrow \langle sc^{\uparrow}(f', M, C), r \rangle \ \text{if} \ f \rightarrow sc^{\uparrow}(f', M, C)$$
$$\text{SITECALL}^* : M(C) \rightarrow sc^{\uparrow}(\gamma, M, C)$$

with γ a constant expression representing a temporary place holder expression. A site call subterm of an expression f rewrites to an operator $(sc^{\uparrow})$ that propagates the call to the root of f using, among others[4], the following equations,

$$sc^{\uparrow}(f_1, M, C) \mid f_2 = sc^{\uparrow}(f_1 \mid f_2, M, C) \ \text{if} \ f_2 \neq \mathbf{0},$$
$$sc^{\uparrow}(f_1, M, C) > x > f_2 = sc^{\uparrow}(f_1 > x > f_2, M, C),$$
$$sc^{\uparrow}(f_1, M, C) \ \textbf{where} \ x :\in f_2 = sc^{\uparrow}(f_1 \ \textbf{where} \ x :\in f_2, M, C),$$
$$f_2 \ \textbf{where} \ x :\in sc^{\uparrow}(f_1, M, C) = sc^{\uparrow}(f_2 \ \textbf{where} \ x :\in f_1, M, C).$$

Once the root of the expression is reached, the effect of the call is reflected in the containing configuration, using the following equation,

$$\langle sc^{\uparrow}(f, M, C), tr : t \mid msg : \rho \mid hdl : h_n \mid clk : c_m \mid r \rangle$$
$$= \langle sc^{\downarrow}(f, h_n), tr : t.M\langle C, h_n \mid m \rangle \mid msg : \rho[M, C, h_n] \mid hdl : h_{n+1} | clk : c_m | r \rangle$$

The effect comprises: (i) the emission of a message $[M, C, h_n]$ to the message pool; (ii) recording of a site call event $M\langle C, h_n | m \rangle$ in the trace (where m is the time at which the event occurs); (iii) updating the handle counter for the next site call; and (iv) replacing the original expression $sc^{\uparrow}(f, M, C)$ by the expression $sc^{\downarrow}(f, h_n)$. Since the handle h_n needs to propagate back to the subterm where the site call was made (which was temporarily substituted by the expression γ), $sc^{\uparrow}(f, M, C)$ does not rewrite immediately to f, but rather to an operator, $sc^{\downarrow}$, that traverses down the expression tree until it reaches the appropriate subterm

---

[4] Concurrent execution of site calls, expression calls and publishing of values is avoided by equations that will introduce an *error* expression in such cases, see [2].

where the handle is inserted.

$$sc^{\downarrow}(f_1 \mid f_2, h) = sc^{\downarrow}(f_1, h) \mid sc^{\downarrow}(f_2, h) \text{ if } f_1 \neq \mathbf{0} \wedge f_2 \neq \mathbf{0},$$
$$sc^{\downarrow}(f_1 > x > f_2, h) = sc^{\downarrow}(f_1, h) > x > f_2$$
$$sc^{\downarrow}(f_1 \text{ where } x :\in f_2, h) = sc^{\downarrow}(f_1, h) \text{ where } x :\in sc^{\downarrow}(f_2, h),$$
$$sc^{\downarrow}(M(P), h) = M(P), \quad sc^{\downarrow}(\mathbf{0}, h) = \mathbf{0}, \quad sc^{\downarrow}(!x, h) = !x, \quad sc^{\downarrow}(!c, h) = !c,$$
$$sc^{\downarrow}(E(P), h) = E(P), \quad sc^{\downarrow}(?h', h) = ?h', \quad sc^{\downarrow}(\gamma, h) = ?h.$$

Expression Call. The specification of an expression call is similar to a site call, in that two operators $ec^{\uparrow}$ and $ec^{\downarrow}$ are defined to propagate the call and its effect to and from the enclosing configuration (see [2] for their defining equations). First, an expression call is modeled with the following rewrite rules.

$$\text{DEF} : \langle f, r \rangle \rightarrow \langle ec^{\uparrow}(f', E, P), r \rangle \text{ if } f \rightarrow ec^{\uparrow}(f', E, P)$$
$$\text{DEF}^* : E(P) \rightarrow ec^{\uparrow}(\gamma, E, P)$$

Then, by means of $ec^{\uparrow}$, the call is propagated up the expression tree to the enclosing configuration, where the effect of the call (appending a τ event to the trace) is recorded and the required declaration is accessed. Using call-by-name semantics, the call is replaced with an instance of the body of the corresponding defining equation. The resulting expression is then propagated back to the appropriate subterm, using the $ec^{\downarrow}$ operator.

Publishing a Value. Publishing a value is modeled by the following rewrite rules.

$$\text{PUB} : \langle f, r \rangle \rightarrow \langle pub(f', c), r \rangle \text{ if } f \rightarrow pub(f', c)$$
$$\text{PUB}^{\tau} : \langle f, r \rangle \rightarrow \langle pub^{\tau}(f'), r \rangle \text{ if } f \rightarrow pub^{\tau}(f')$$
$$\text{PUB}^* : !c \rightarrow pub(\mathbf{0}, c)$$

The publishing expression rewrites to an operator pub that replaces it with the zero expression $\mathbf{0}$ and then initiates the process of propagating the published value c up the expression tree (see the equations in [2]). If c is *not* bound in the expression, the value is propagated all the way to the top and a publish event is recorded in the enclosing configuration. Otherwise, if the value published is bound by a sequential composition expression or an asymmetric parallel composition expression, then one of the following equations applies:

$$pub(f, c) > x > g = pub^{\tau}(f > x > g \mid g\{c/x\})$$
$$g \text{ where } x :\in pub(f, c) = pub^{\tau}(g\{c/x\})$$

The equations reflect the semantics specified by the SOS rules SEQ1V and ASYM1V of Figure 3. They also transfer the propagation task to another operator pub^{τ}, which ultimately causes a τ event to be recorded in the trace field of the configuration.

Site Return. Although, the environment in the SOS specifications of Figure 3 is treated as a "black box" with unpredictable responses from remote sites, we need to simulate environment responses in order to arrive at an executable specification. This is achieved in the following way. Once a message $[M, C, h_n]$ is emitted

into the message pool as a result of a site call, it is converted into the message $[self, app(M, C, rand), h_n]$, which represents a *potential* response back to $self$, a reference to the current expression. The operator $app(M, C, rand)$, whose definition depends on the site M, associates a pseudo-random delay to responses from remote sites. Once a response from M is returned, the site return rule may fire.

$$\text{SiteRet} : \langle f, tr : t \mid msg : \rho \, [self, c, h] \mid clk : c_m \mid r \rangle \rightarrow$$
$$\langle sr(f, c, h), tr : (t.h?c|m) \mid msg : \rho \mid clk : c_m \mid r \rangle \text{ if } h \in handles(f)$$

Application of the site return rule is subjected to the condition that the handle name of the message to be consumed is referenced in f. This is to avoid useless transitions that could take place when a thread, having an unfinished site call, is pruned using the **where** statement. If the condition is satisfied, the incoming message is consumed, a site return event $h?c|m$ is generated, and the expression f is replaced with an operator $sr(f, c, h)$ that propagates the return value down the expression tree to the appropriate pending call (see the equations in [2]).

3.3 Asynchronous versus Synchronous Execution Strategies

The rewrite theory described above does not enforce any execution strategy among instantaneous transitions of an Orc configuration as it allows internal transitions within an Orc expression (site calls, expression calls, and publishing of values) and the external transition of interacting with sites (site returns) to be interleaved asynchronously in any order. It reflects the exact behavior of the SOS semantics specification of Figure 3, which is in some sense too loose. In particular, site returns may take place in an expression while site calls that are ready to be made are waiting. For example, in the expression $DelayedN \mid M$, the call to M may be delayed, thus defeating the purpose of prioritizing the call to M. This issue was discussed in [22], where a *synchronous* semantics is proposed by placing further constraints on the application of SOS semantic rules of Figure 3. The synchronous semantics is arrived at by distinguishing between *internal* and *external* events, and splitting the SOS transition relation $\hookrightarrow$ into two subrelations $\hookrightarrow_R$, and $\hookrightarrow_A$, and characterizing set-theoretically, the complementary subsets of expressions (*quiescent* vs. *non-quiescent*) to which they are respectively applied.

In the context of rewriting logic, we showed in [1] how this restriction can be captured precisely in the SOS-based rewriting semantics using two alternative approaches: (1) *strategy expressions* [11]; and (2) *equationally defined predicates*. Although these two approaches are readily applicable to the rewrite theory developed here, we focus our attention in this paper on the latter approach, because strategy expressions are, as of this writing, not yet fully supported in Maude. We obtain a synchronous reduction semantics of Orc by means of a more precise version of $\mathcal{R}_{Orc}^{red}$ that gives the site return rule the lowest priority among the instantaneous actions. We first introduce the notion of an *active expression*.

Definition 3.1 The set of *active* expressions f_{active} in $\mathcal{R}_{Orc}^{red}$ is the smallest set generated by the following rules.

(i) $M(C)$, $E(P)$, and $!c$ are all in f_{active}.

(ii) $f \mid g \in f_{active}$ if $f \in f_{active}$ or $g \in f_{active}$.

(iii) $f > x > g \in f_{active}$ if $f \in f_{active}$.

(iv) g **where** $x :\in f \in f_{active}$ if $f \in f_{active}$ or $g \in f_{active}$.

Note that our notion of an active expression exactly corresponds to the non-quiescent expression in [22]. This notion can be easily equationally captured by a predicate $active : Expr \rightarrow$ [Bool] [frozen] (see [2] for its defining equations), which is then used to limit the application of the SITERET rules, as follows

$$\text{SITERET} : \langle f, tr : t \mid msg : \rho[self, c, h_n] \mid clk : c_m \mid r \rangle \rightarrow$$
$$\langle sr(f, c, h_n), tr : (t.h_n?c|m) \mid msg : \rho \mid clk : c_m \mid r \rangle$$
$$\text{if } h_n \in handles(f) \wedge active(f) \neq true$$

3.4 Timed Semantics

An important aspect of Orc is that of time elapse. Transitions of an Orc program, such as $DelaydN$, may occur at different times. Moreover, responses from non-local sites, such as CNN and BBC, may experience unpredictable delays and communication failures, which are inherent in Orc's computation model. This is in contrast to local sites, such as $atimer$ and $rtimer$, for which Orc provides strong temporal guarantees. In order to be able to reason about real-time guarantees of Orc programs, time elapse needs to be modeled explicitly. For this purpose, as usual for rewriting logic semantic definitions of real-time systems [24], we use a (discrete) time domain (maintained by the *clock* field in a configuration), and an additional rewrite rule, the "tick" rule, to advance time: $\langle f, clk : c_m \mid r \rangle \rightarrow \langle f, clk : c_{m+1} \mid \delta(r) \rangle$, where δ is a function that updates the record r in the state of the configuration to reflect the elapse of one time unit. However, as explained in [1], the addition of the tick rule may introduce uninteresting or undesirable behaviors. For instance, an Orc configuration that could make an instantaneous transition might instead choose to keep advancing time indefinitely without making any real progress. This should be avoided by giving time-elapsing rewrites the *lowest possible priority*. That is, we need to define a *time-synchronous* execution semantics, in which a configuration is not allowed to advance its time unless it reaches a state where no internal or external action, other than a time tick, can be taken [5]. Therefore, the theory $\mathcal{R}_{Orc}^{red}$ specifying the synchronous semantics of Orc is further extended in the following way. We first define an *eager* configuration as one that can make an instantaneous action.

Definition 3.2 An Orc configuration $\mathcal{C}$ in $\mathcal{R}_{Orc}^{red}$ is *eager* if $\mathcal{C}$ is of one of the following forms: (i) $\langle f, r \rangle$ with $f \in f_{active}$; or (ii) $\langle f, msg : \rho [self, c, h_n] \mid r \rangle$ if $h \in handles(f)$.

This notion of eager configurations can be easily captured by a predicate *eager* : $Conf \rightarrow$ [Bool] [frozen], which evaluates to *true* if and only if it is applied to a

---

[5] This time-synchronous strategy has a limitation, for a discussion of which the reader is referred to [1].

configuration that can make an instantaneous action (see [2] for the equations of *eager*). To capture the desired time-synchronous semantics in $\mathcal{R}_{Orc}^{red}$, we restrict the application of the tick rule by the condition that the configuration is not eager.

$$\text{TICK} : \langle f, clk : c_m \mid r \rangle \rightarrow \langle f, clk : c_{m+1} \mid \delta(r) \rangle \text{ if } eager(\langle f, clk : c_m \mid r \rangle) \neq true$$

3.5   Correctness of the Semantics

In order for the rewriting semantics specifications described above to capture the intended semantics of Orc, it must somehow correspond to the SOS specifications of Figure 3. In this section, we present an equivalence theorem of which a detailed discussion and a complete proof are given in [2]. The theorem entails that the rewriting semantics of Orc given by $\mathcal{R}_{Orc}^{red}$ and the SOS-based rewriting semantics $\mathcal{R}_{Orc}^{sos}$ developed in [1] are semantically equivalent, in the sense that an Orc program behaves in exactly the same way in both semantic models. The correctness of $\mathcal{R}_{Orc}^{red}$ against the synchronous SOS semantics of [22] then follows immediately from that of $\mathcal{R}_{Orc}^{sos}$ with respect to the same SOS semantics, which was studied in [1].

Definition 3.3 *In a configuration $\langle f, (\text{con} : \sigma) \mid r \rangle$, an occurrence of an expression name E is* bound *in f if there exists a declaration for E in the context σ. Otherwise, E is said to be* free *in f. Likewise, an occurrence of E is* bound *in σ if there exists a declaration for E in σ, and is* free *in σ otherwise.*

Definition 3.4 *An Orc configuration $\langle f, r \rangle$ is* well-formed *if: (1) f does not contain any of the auxiliary function symbols introduced in Section 3.2, such as $sc^{\uparrow}$, pub, and γ; and (2) r contains at least the five fields listed in Section 3.1. Moreover, a* closed *configuration is a well-formed configuration in which no expression name appears free in f or σ, the context component of r.*

Theorem 3.5 *[2] For any closed configurations $\mathcal{C}$ and $\mathcal{C}'$, the following equivalence holds: $\mathcal{C} \rightarrow_{\mathcal{R}_{Orc}^{sos}} \mathcal{C}' \Longleftrightarrow \mathcal{C} \rightarrow_{\mathcal{R}_{Orc}^{red}} \mathcal{C}'$.*

Therefore, for any Orc program P, the state transition systems defined by $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ are strongly bisimilar.

3.6   Performance Comparison

Although the two theories $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ are semantically equivalent, $\mathcal{R}_{Orc}^{red}$ is much more efficiently executable and analyzable. To validate this claim, both theories were implemented in Maude and a number of experiments were conducted using some Orc programs that originally appeared in [22]. In all experiments, performance was measured in terms of the time taken to perform a particular task. The tasks were: (1) simulating four Orc programs using Maude's `rewrite` command; (2) exploring the state space of these four programs using Maude's breadth-first `search` command; and (3) model checking three instances of the dining philosophers problem using Maude's LTL model checker [6]. The results of these experiments are

---

[6] For the first two tasks, the clock was limited to ten clock ticks, and pseudo-random delays were assumed. Since the model checking task did not require external communication, time in this case was limited to a

		TIMED-MCALL	TIMEOUT	PRIORITY	PARALLEL-OR	DINING PHILOSOPHERS		
						2	3	4
$\mathcal{R}_{Orc}^{sos}$	rewrite	2,178	27	47	3,247	213	51,268	∞
	search	∞	376	1,921	143,158			
$\mathcal{R}_{Orc}^{red}$	rewrite	4	2	2	3	53	1,502	45,860
	search	34,396	31	188	4,861			

Table 1
A performance comparison of the rewriting semantics of Orc. A number in the table represents the CPU time in milliseconds, as reported by Maude, to finish the corresponding task.

summarized in Table 1, which clearly shows that the reduction semantics can be executed and analyzed much more efficiently than the SOS-based semantics.

4 Distributed Object-based Rewriting Orc Semantics

Many orchestration applications, especially relatively large ones, can be thought of as consisting of multiple Orc subexpressions independently orchestrating different but related tasks. For instance, in the dining philosophers implementation in Orc [22] with n philosophers, there are n subexpressions running in parallel, one for each philosopher. In more practical applications, such subexpressions normally run on physically distributed autonomous agents spread across the web. Furthermore, sites, whose responses were only simulated in the rewriting semantics developed in the previous section to arrive at an executable specification, normally maintain local states to support the services they provide, such as counter sites and channel (buffer) sites. Therefore, it is natural to think of Orc expressions and sites as *objects* in a distributed configuration. Expression objects are active objects (or actors in the actor model) having a state and one or more threads of control, and are capable of initiating (asynchronous) message exchange. Site objects are reactive objects having internal states and are capable only of responding to incoming requests. They can be thought of as actors that have a passive-reactive behavior.

The reduction semantics described in Section 3 is a key step towards the specification of the object-based semantics of the Orc's orchestration model. In addition, within the Maude framework, the object-based semantics lends itself nicely to a future (physically) distributed deployment using Maude's socket programming capabilities. This leads to a formal analysis and verification environment that is faithful to the distributed nature of Orc's wide-area computations.

4.1 Distributed Orc Semantics

A *distributed Orc configuration* is modeled by a multiset of objects and messages. There are three classes of objects, namely, *expression*, *site*, and *clock* objects. a *clock* object is a simple object of class *Clock*, which maintains a single field, called *clk*, representing the current clock time. An *expression* object is an object of class *Expr* having three attributes: (i) *exp*, which holds an Orc expression to be computed;

---

single clock tick with no delays. All experiments were run on a 3.2GHz dual-core machine with 2GB of memory using Maude 2.3.

(ii) *con*, which is the context where expression name declarations appear; and (iii) *hdl*, which maintains a set of handle names that are currently being used by the expression. A *site* object is one of class *Site* with the following attributes:

(i) *name*: the name of the site, such as *if*, *rtimer*, *CNN*, *BBC*, . . . etc.

(ii) *op*: the current operation being performed by the site. This attribute indicates whether the site object is currently blocking or accepting incoming messages. It also serves as a means to modularly specify a particular site definition.

(iii) *state*: the processing state of a site object. This field is abstractly defined as a list of items whose concrete meaning depends on the particular site being specified. Fundamental sites, such as *if* and *rtimer*, and other basic sites, such as arithmetic functions, are stateless and thus make no use of this field. However, more complex sites may require this attribute to maintain their state.

A *message* is either a *site call* message of the form $M \leftarrow sc(O, C, h, m)$, with M the name of the site being called, and O the object identifier of the caller expression object, or a *site return* message of the form $O \leftarrow sr(c, h, m)$, with O the identifier of the expression object receiving the published value c.

The distributed semantics of an Orc expression object is essentially that of the reduction semantics specification of Section 3, with the exception that messages are now managed by the distributed Orc configuration. This distributed semantics generalizes the reduction semantics to multiple Orc expressions, and provides an explicit treatment of message exchange between expression and site objects. A detailed discussion of this distributed semantics and its implementation in Maude can be found in [2]. We illustrate through an application how Maude's LTL model checker can be used to verify properties of distributed Orc systems.

4.2 A Case Study: Managing an Online Auction

The distributed Orc auction program AUCTION presented here was inspired by the Orc auction example given in [22]. The program uses a few expression declarations that we briefly describe first (a detailed discussion of the program and the following analysis can be found in [2]). The two main declarations are *PostingDecl* and *BiddingDecl*. *PostingDecl* defines an expression *Posting(S)* that gets items that are available to be advertised from the seller site list S, which is given below [7].

$$Posting(S) =_{def} if(empty(S)) \gg let(0) \mid if(\neg empty(S)) \gg$$
$$(S_0(PostNext) > item > auction(post, item) \gg rtimer(item_1 + 1)$$
$$\gg Posting(tail(S)))$$

An item is a tuple (id, t, m), with id the item's identifier, t the duration of the auction, and m the minimum bid. Once an item is posted, the expression waits for

---

[7] Subscripts are used to denote zero-based indexing of elements in a list. For example, S_0 is the first element in S and $item_1$ is the second element in $item$. Furthermore, the notation $\gg$ is used for sequential composition when no value passing occurs.

the auction to end before proceeding to the next item. The declaration *BiddingDecl* defines the bidding expression that manages the bidding process and announces winning bidders.

$$Bidding(B) =_{def} auction(getNext) > item > Bids(item_0, item_1, item_2, B, 0) > w >$$

$$(if(w_1 = 0) \gg Bidding(B) \mid if(w_1 \neq 0) \gg w_1(won, item_0, w_0))$$

B is a list of bidders and *Bids* is an expression, declared by *BidsDecl* shown below, which, if successful, returns a pair $(wbid, wbidder)$ consisting of the winning bid and the winning bidder name.

$$Bids(id, duration, bid, B, winner) =_{def} if(duration = 0) \gg let(bid, winner)$$

$$\mid if(duration \neq 0) \gg Collect(nextBid, B, id, bid) > bidList >$$

$$MaxBid(bidList) > m > rtimer(1) \gg Bids(id, duration - 1, m_0, B, m_1)$$

The *Bids* expression collects bids in rounds, each lasting for one time unit. In each round, the maximum bid is computed and published by the site *MaxBid*, and then used as the minimum bid for the next round. The *Collect* expression (declared by *CollectDecl* shown below) returns a list of bidding pairs of the form $(bid, bidder)$.

$$Collect(m, B, id, minBid) =_{def} if(empty(B)) \gg let(nil) \mid if(\neg empty(B)) \gg$$

$$(append(x, xs) \textbf{ where } x :\in B_0(m, id, minBid)$$

$$\textbf{where } xs :\in Collect(m, tail(B), id, minBid))$$

Beside the clock object and the fundamental site objects, the initial configuration of AUCTION used in this section contains two expression objects: the posting expression object and the bidding expression object,

$$\langle o_1 : Expr \mid exp : Posting(seller_0), \; con : PostingDecl, \; hdl : \emptyset \rangle$$
$$\langle o_2 : Expr \mid exp : Bidding(b_0, b_1, b_2), \; con : BiddingDecl, BidsDecl, CollectDecl, \; hdl : \emptyset \rangle$$

along with six site objects: (1) a seller site object whose name is $seller_0$; (2) three bidder site objects (named b_0, b_1, and b_2); (3) a site object for the *auction* site, which manages the bidding process; and (4) a site object for the *MaxBid* site. In AUCTION, we assume that $seller_0$ has two items t_1 and t_2 for sale and that bidders follow different bidding strategies. For simplicity, we assume no communication delays, and allow enough clock ticks for program completion.

We can specify some correctness properties of AUCTION, and then verify them using Maude's LTL Model Checker. Four atomic predicates, which are parametric to items, are used. $hasbid(t)$ and $sold(t)$ are self-explanatory. $max(t)$ is true in a state where t is sold to the highest bidder, while $conflict(t)$ is true whenever t has two or more winning bidders.

(i) An item with at least one bid is eventually sold: $\Box \bigwedge_i (hasbid(t_i) \rightarrow \Diamond sold(t_i))$

```
Maude> red modelCheck(init, commitAll) .
rewrites: 33366205 in 59315ms cpu (59332ms real) (562516 rewrites/second)
result Bool: true
```

(ii) An item is always sold at the maximum bid to the highest bidder: $\Box \bigwedge_i (sold(t_i) \rightarrow max(t_i))$

```
Maude> red modelCheck(init, winAll) .
rewrites: 33739349 in 61027ms cpu (61024ms real) (552852 rewrites/second)
result Bool: true
```

(iii) An item cannot have two winners: $\neg \Diamond \bigvee_i conflict(t_i)$

```
Maude> red modelCheck(init, uniqueWinnerAll) .
rewrites: 33290882 in 59742ms cpu (59739ms real) (557235 rewrites/second)
result Bool: true
```

5 Conclusion and Future Work

We have presented an efficient reduction semantics for Orc, shown how it captures Orc's synchronous real-time semantics, and established its semantic equivalence with a previous SOS-like semantics. We have also further refined the reduction semantics into a distributed object semantics and have shown how LTL properties of Orc programs can be model checked using the distributed semantics. A natural future extension of this work is the development of a provably correct distributed implementation of Orc. The key idea is to shift the emphasis in the use of rewriting logic from executable *specification* to declarative distributed *programming*. In particular, we expect to make heavy use of Maude's support for sockets as external objects [9] to develop such a distributed implementation.

Acknowledgement

References

[1] Musab AlTurki and José Meseguer. Real-time rewriting semantics of Orc. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming*, pages 131–142, New York, NY, USA, 2007. ACM Press.

[2] Musab AlTurki and José Meseguer. Rewriting logic semantics of Orc. Technical Report UIUCDCS-R-2007-2918, University of Illinois at Urbana Champaign, November 2007.

[3] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[4] J. C. M. Baeten and Cornelis A. Middelburg. *Process algebra with timing; Monographs in theoretical computer science.* Springer, Berlin; New York, 2002.

[5] Roberto Bruni, Hernán Melgratti, and Emilio Tuosto. Translating Orc features into petri nets and the join calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2006.

[6] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1-3):386–414, 2006.

[7] Liang Chen. An interleaving model for real-time systems. In *TVER '92: Proceedings of the Second International Symposium on Logical Foundations of Computer Science*, pages 81–92, London, UK, 1992. Springer-Verlag.

[8] Shuo Chen, José Meseguer, Ralf Sasse, Helen J. Wang, and Yi-Min Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy*, pages 71–85. IEEE Computer Society, 2007.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.3). January 2007. http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf.

[11] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. *Electron. Notes Theor. Comput. Sci.*, 174(11):3–25, 2007.

[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 *Proceedings of the NATO Advanced Study Institute on Engineering, Theories of Software Intensive Systems*, Marktoberdorf, Germany, August 2004.

[14] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. *CONCUR 2006 –Concurrency Theory*, pages 477–491, 2006.

[15] J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR'04, Cork, Ireland, July 2004*, pages 1–44. Springer LNAI 3097, 2004.

[16] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[17] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[18] José Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.

[19] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. *Algebraic Methodology and Software Technology*, pages 364–378, 2004.

[20] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *Theor. Comput. Sci.*, 373(3):213–237, 2007.

[21] Jayadev Misra. Computation orchestration: A basis for wide-area computing. In Manfred Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, Germany, 2004.

[22] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006.

[23] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.

[24] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[25] Sidney Rosario, David Kitchin, Albert Benveniste, William Cook, Stefan Haar, and Claude Jard. Event structure semantics of Orc. In *4th International Workshop on Web Services and Formal Methods (WS-FM 2007)*, Brisbane, Australia, October 2007.

[26] Steve Schneider, Jim Davies, D. M. Jackson, George M. Reed, Joy N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 640–675, London, UK, 1992. Springer-Verlag.

[27] Ian Wehrman, David Kitchin, William R. Cook, and Jayadev Misra. A timed semantics of Orc. *Theoretical Computer Science*, July 2007. To appear (preliminary version in http://www.cs.utexas.edu/users/wcook/Drafts/2007/TimedSemanticsDRAFT.pdf).