

Real-Time Rewriting Semantics of Orc

Musab AlTurki José Meseguer

University of Illinois at Urbana-Champaign
201 N Goodwin Ave
Urbana, IL 61801, USA
{alturki,meseguer}@cs.uiuc.edu

Abstract

Orc is a language proposed by Jayadev Misra [19] for *orchestration* of distributed services. Orc is very simple and elegant, based on a few basic constructs, and allows succinct and understandable programming of sophisticated applications. However, because of its real-time nature and the different priorities given to internal and external events in an Orc program, giving a formal operational semantics that captures the real-time behavior of Orc programs is nontrivial and poses some interesting challenges. In this paper we propose such a real-time operational Orc semantics, that captures the informal operational semantics given in [19]. This operational semantics is given as a rewrite theory in which the elapse of time is explicitly modeled. The priorities between internal and external events are also modeled in two alternative ways: (i) by a rewrite strategy; and (ii) by adding extra conditions to the semantic rules. Since rewriting logic has efficient implementations such as Maude, we also get, directly out of the semantic definitions, both an Orc interpreter and an LTL model checker for Orc programs.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Operational semantics; Program analysis

General Terms Languages, Theory, Verification

Keywords Orc, rewriting logic, structural operational semantics, real-time, orchestration theory, formal analysis, Maude

1. Introduction

Orc is a language proposed by Jayadev Misra [19] for *orchestration* of distributed services. The language is very simple and elegant, based on a few basic constructs, and allows succinct and understandable programming of sophisticated applications, including web services and their coordination. However, although the language is conceptually simple, giving it a precise formal semantics accounting for all its aspects is nontrivial and poses some interesting challenges, which we address in this paper.

Several Orc semantics have already been given. Hoare, Menzel and Misra [12] developed a tree-based denotational semantics of Orc that is well-suited for reasoning about identities in the language, rather than for describing the operational behavior of

Orc programs. Another formal semantic model intended mainly to prove algebraic laws about Orc expressions appeared in [13], where the authors proposed a trace-based characterization of Orc expressions. We are particularly interested in Orc's *operational* semantics. A precise but informal operational semantics for Orc was given by Misra in [19]; 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 [20]; 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 [20].

We take the asynchronous and synchronous formal operational semantics of Orc given in [20] as our basis. However, in our view the intended informal operational semantics in [19] is not yet fully captured. The key point is that Orc is a *real-time* programming language, in which time elapse is of the very essence for an Orc computation. But time elapse is not explicitly modeled in either the asynchronous or synchronous semantics in [20]. Modeling time elapse explicitly is important for reasoning about real-time guarantees of Orc programs. An Orc program may call various *non-local* sites (for example, BBC or CNN news sites), and then no time guarantees can be given: the sites may give delayed responses or may fail to respond. But an Orc program may also call *local* sites, such as timers, for which very strong real-time guarantees can be given. Our goal in this paper is to give a formal *real-time* operational semantics of Orc that supports reasoning and automated verification about such real-time properties of Orc programs.

Providing such a real-time operational semantics poses a number of technical challenges, which we tackle using the rewriting logic approach to programming language semantics (see [17, 18] and references there). The easy part is representing the SOS rules of the asynchronous semantics given in [20]. For this, we use a general technique, developed in [16] and summarized in Section 2.2, to faithfully map Modular Structural Operational Semantics (MSOS) definitions in the sense of Mosses [21] to a corresponding rewrite theory. We do exactly this for Orc's asynchronous SOS semantics in Section 4.2. The first nontrivial challenge is to capture in our framework the Orc *synchronous* operational semantics given in [20] as a restriction on the application of the asynchronous SOS rules. This first challenge can be met in two alternative ways: (i) by restricting the application of the corresponding rewrite rules using the *strategy language* proposed in [10] and summarized in Section 2.4; and (ii) by adding further *conditions* to the semantic rules, thus making the use of strategies unnecessary. We describe both approaches in this paper. The synchronous Orc semantics based on method (i) is given in Section 4.3; and a real-time semantics extending the synchronous semantics and based on method (ii) is sketched in Section 5.1. The second nontrivial challenge is to give a real-time formal operational semantics. For this we use the gen-

eral method to represent real-time systems as rewrite theories first proposed in [23] and supported by the Real-Time Maude tool [24]. Our real-time rewriting logic semantics for Orc is given in Section 5. But that leaves still open a third challenge, which is to give a *synchronous* real-time formal semantics for Orc, in which internal and external actions are given priority over time elapse. For this, the two methods (i)–(ii) mentioned above can be used. Since each has its advantages, we describe the application of both methods in Section 5.1.

A pleasant side-effect of meeting all the above challenges is that, since rewriting logic is a computational logic [14], which has high-performance implementations supporting both execution and automated verification such as Maude [8], we obtain *for free*: (i) an Orc interpreter; and (ii) an Orc LTL model checker in which we can verify safety and liveness properties of Orc programs. Section 6 illustrates the use of the Orc interpreter and the formal analysis of Orc programs supported by the Orc model checker, which we have obtained directly out of the real-time rewriting semantics of Orc specified in Maude. We finish the paper with some concluding remarks in Section 7.

2. Preliminaries

2.1 Rewriting Logic as a Semantic Framework

Rewriting logic [14] 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 [17, 18] and references there), and to model real-time systems [23]. Furthermore, with the availability of high-performance rewriting logic implementations, such as Maude [8], 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 [15], 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:

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

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

- $\phi : \Sigma \rightarrow \mathcal{P}(\mathbf{N})$ a function that assigns to each operator symbol f in Σ of arity $n > 0$ a set of positive integers $\phi(f) \subseteq \{1, \dots, n\}$ 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 [5]. Using these inference rules, a rewrite theory \mathcal{R} proves a statement of the form $(\forall X) t \rightarrow t'$, written as $\mathcal{R} \vdash (\forall X) t \rightarrow t'$, 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 [14]. [5] gives a precise account of the most general form of rewrite theories and their models.

2.2 MSOS to Rewriting Logic Transformation

Modular structural operational semantics (MSOS) [21] specifications can be naturally mapped to semantically equivalent rewrite theories in rewriting logic. In general, a rule in MSOS corresponds to a conditional rewrite rule in rewriting logic. Meseguer and Braga [16] described a semantics-preserving transformation from MSOS to rewriting logic that results in modular rewrite theories and accounts for the single-step MSOS rules. Given an MSOS specification of the semantics of a programming language \mathcal{L} , the transformation uses a pair $\langle P, R \rangle$, called a *configuration*, where P is a program text in \mathcal{L} , and R is a record consisting of fields that contain state information necessary for the semantics of P , such as environments, stores, and traces.

The transformation also describes a method of controlling the number of rewrites in the condition of a rewrite rule. This is required in such a transformation because SOS transitions are single-step, whereas sequents in rewriting logic can involve an arbitrary (but finite) number of steps because of the reflexivity and the transitivity inference rules of the logic. First, assuming P is a program expression and R is a record, to achieve the one-step SOS behavior, two more syntactic forms of a configuration $\langle P, R \rangle$ are defined: $[P, R]$ and $\{P, R\}$. Then, all the semantic definitions are specified using rewrite rules of the form

$$\langle P, R \rangle \rightarrow \langle P', R' \rangle \text{ if } \bigwedge_{i=1}^n \{P_i, R_i\} \rightarrow [P'_i, R'_i] \wedge C$$

where C is (a possibly empty) conjunction of (conditional) equations and/or memberships. Now a one-step rewrite of $\langle P, R \rangle$ is achieved by the following rewrite rule.

$$[\text{STEP}] : \langle P, R \rangle \rightarrow \langle P', R' \rangle \text{ if } \{P, R\} \rightarrow [P', R']$$

The reader is referred to [16] for a more detailed discussion of the methodology and a proof of its semantics-preserving correctness.

2.3 The Maude System

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) . Functional modules are declared with the syntax `fmod <name> is <body> endfm`, where `<name>` is a name given to the module and `<body>` consists of module inclusion assertions, sort and subsort declarations, operator symbols declarations, and (possibly conditional) equations and membership axioms. System modules, which are declared with the `mod . . . endm` keywords, may additionally contain (possibly conditional) rewrite rules.

Maude provides several features and tools to formally analyze specifications given as system modules. The features include: (1) the `rewrite` command (abbreviated as `rew`), which applies in a fair manner the rules, equations, and membership axioms in the system module on a given term, resulting in a sample run of the program specified by the module, and (2) the `search` command, which performs a breadth-first search on the states reachable from a given state while looking for states matching a given pattern and satisfying a semantic condition. In effect, the `search` command provides a semi-decision procedure for checking violations of invariants. Maude also provides an LTL model checker to verify more complex LTL safety and liveness properties about finite state systems. The use of these tools for analyzing Orc programs is illustrated in Section 6. For a complete description of these features and tools, and various other tools provided by Maude, the reader is referred to the Maude book [7] and the Maude manual [8].

L	\in	Labels
R	\in	Strategy names
$B \in$ Basic Strategy	$::=$	$\text{idle} \mid \text{fail} \mid L \mid L[SL]$
$S, S_1, S_2 \in$ Strategy	$::=$	$R \mid B \mid S_1; S_2 \mid S_1 \mid S_2$ $\mid S^* \mid S^+ \mid S? \mid S_1 : S_2$
$SL \in$ Strategy List	$::=$	$S \mid S, SL$
$D \in$ Strategy Declaration	$::=$	$R := S$
$DL \in$ Strategy Specification	$::=$	$D \mid D, DL$

Table 1. The syntax of a subset of Maude’s strategy language

2.4 The Maude Strategy Language

Maude’s strategy language [10] is a relatively simple language in which strategy expressions specify how terms in a rewrite theory are rewritten. The goal of the strategy language is to provide a means of controlling the application of rules in a rewrite theory while keeping these control mechanisms separate from the system specifications given by that theory. The strategy language of Maude achieves this separation by having *strategy modules* specifying strategy expressions that are distinct from system modules, which specify rewrite theories. Set-theoretically, the meaning of a strategy expression S is a function that, when applied to a term t , yields a (possibly empty) set of terms, which are the terms that can be obtained by applying this strategy.

$$_@_ : \text{Strat} \times T_\Sigma(X) \rightarrow \mathcal{P}(T_\Sigma(X))$$

This function is extended to sets of terms, in the obvious way, so that if $T \in \mathcal{P}(T_\Sigma(X))$, then $S@T = \bigcup_{t \in T} S@t$.

Besides providing basic strategies through the use of rule labels, the strategy language permits combining these strategies into more complex ones using several combinators. Furthermore, substrategies may be specified for rewrite conditions of a rewrite rule. In the rest of this section, only the subset of the strategy language that is most relevant for this work is described. For a detailed discussion of the entire language, the reader is referred to [10].

Table 1 shows the syntax of a subset of the strategy language. The simplest strategies are *idle*, which does not affect the term to which it is applied (i.e., $\text{idle}@t = \{t\}$), and *fail*, which always gives the empty set as its result ($\text{fail}@t = \emptyset$). A basic strategy can also be a label of a rule, which when applied to a term results in the set of terms obtained by applying the rule to t . For a conditional rewrite rule with n rewrite conditions, the label may optionally be followed by a list of n strategy expressions controlling the way the rewrite conditions are checked. If no such list is given, the conditions are not restricted to any strategy and Maude’s breadth-first search is applied to evaluate conditions.

Strategy expressions may be combined using regular expression combinators: concatenation (\cdot), union (\mid), and iteration (S^* for zero or more iterations and S^+ for one or more iterations). Additionally, there is the generalized conditional combinator $S? S_1 : S_2$, which, when applied to a term t , behaves as follows. S is first applied to t resulting in a set T . If S succeeds (i.e. $T \neq \emptyset$), then the result of S_1 applied to T is returned. Otherwise, if S fails ($T = \emptyset$), then the result of applying S_2 to the original term t is returned. Other derived strategies may be defined using these combinators [10].

Finally, strategy expressions can be given names through strategy declarations of the form $R := S$. This allows defining (mutually) recursive strategies, in addition to modularly decomposing large strategy expressions into smaller ones. A strategy specification is simply a list of strategy declarations.

$\text{let}(x, y, \dots)$	A tuple constructor. Given a list of values, it returns a tuple consisting of these values.
<i>clock</i>	returns the current time as a non-negative integer.
<i>atimer</i> (t)	returns a signal at time t .
<i>rtimer</i> (t)	returns a signal after t time units.
<i>signal</i>	returns a signal immediately.
<i>if</i> (b)	returns a signal if b is true; otherwise it remains silent.

Table 2. Fundamental sites in Orc

3. Orc and its Semantics

Orc is a theory of orchestration that models the smooth integration of web services. It is based on the abstract notion of sites and the composition of the services they provide. The Orc model is fairly minimal, yet powerful enough to express a wide range of computations [20]. Furthermore, Orc assumes a timed framework in which services and object states may be time-sensitive.

A central concept in Orc is that of sites. A *site* is a basic service that provides a computation of some kind. 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 . Finally, 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 Table 2, assuming t is a positive integer, b is a boolean, and x and y are values of arbitrary types.

Complex *expressions* in Orc are built from smaller ones using three composition operators: (1) the sequential composition operator ($\langle x \rangle$), where x is a variable; (2) the symmetric parallel composition operator (\mid), which expresses parallel threads of computation; and (3) the asymmetric parallel composition operator (*where*), which expresses a form of parallelism where some concurrent threads of computation can be selectively removed at some stage in their execution. An Orc expression may, therefore, return as its result a sequence of values of possibly different types, or it may not return a value at all.

3.1 Orc Syntax and Informal Semantics

We adopt a variant of the syntax of Orc that encapsulates Orc declarations and expressions into a single unit (an Orc program). This variant originally appeared in an earlier version of [9]. The extended Orc syntax we use here is given in Table 3.¹ An Orc *program* consists of an optional list of declarations followed by an Orc expression. A *declaration* is similar to a procedure declaration, in that it consists of a name, a (possibly empty) list of formal parameters, and an expression representing the body of the procedure. 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 (more on this later); or (5) the composition of two expressions by one of the three composition operators. Two expressions may be composed either

¹In this syntax, we allow polyadic communications in site calls and polyadic expression declarations and calls. We also extend the original syntax with expressions that will be needed for defining the semantics later on. The syntax, however, treats site names and variables as two separate entities, implying that a site cannot be bound in an expression or be used as a formal parameter in a declaration, although a site may be used as an actual parameter in a site or an expression call.

sequentially with the sequencing operator $> x >$, or in parallel. Parallel composition comes in two flavors: (1) symmetric composition, using $|$, 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. The formal asynchronous semantics using SOS specifications is given in Section 3.3. Below we give some examples illustrating the informal semantics.

3.2 Examples

To illustrate the behavior of the different composition operators, we describe a few example Orc expressions, which can be found, among many other examples, in [19, 20]. We will refer to some of these expressions later in the paper. Also, to cut down on the use of parentheses, we assume that the composition operators are ordered in decreasing precedence as follows: $> x >$, $|$, **where**. We also let $> x >$ and **where** be right associative, and $|$ be commutative and fully associative.

The following is an expression (which we call **TIMED-MCALL**) that calls site M four times, in intervals of one time unit each, starting immediately.

$$\begin{array}{l} M \quad | \quad \text{rtimer}(1) > x > M \\ \quad \quad | \quad \text{rtimer}(2) > x > M \\ \quad \quad | \quad \text{rtimer}(3) > x > M \end{array}$$

The program **TIMEOUT** below encodes a form of timeout. It consists of a declaration of an expression f that has an input parameter t representing the timeout, and an expression call setting the timeout to 3.

$$f(t) =_{def} \text{let}(z) \text{ where } z : \in M \mid \text{rtimer}(t) > x > \text{let}(0); f(3)$$

In the call $f(3)$, t is bound to 3 and a call to site M is made. If M responds before 3 time units, then its value is the value published by the expression, while the value 0 is published if no response is received after 3 time units have elapsed. If M responds exactly after 3 time units, either value is published.

Program **PRIORITY** below implements a prioritized site call:

$$\begin{array}{l} \text{DelayedN} =_{def} \text{rtimer}(1) > x > (\text{let}(u) \text{ where } u : \in N); \\ \text{let}(x) \text{ where } x : \in M \mid \text{DelayedN} \end{array}$$

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.

The last example we present here illustrates a simple recursive program. The following declares an expression that recursively publishes a signal every time unit, indefinitely.

$$\text{Metronome} =_{def} \text{signal} \mid \text{rtimer}(1) > x > \text{Metronome}$$

The expression *Metronome* can be used to repeatedly initiate an instance of a task every time unit.

3.3 Asynchronous Structural Operational Semantics of Orc

The asynchronous operational semantics introduced in [20] (and shown here in Figure 1 below) formalizes the general description of the meanings of the various Orc features given above. The SOS semantics is a highly non-deterministic semantics that allows internal transitions (within an Orc expression) and external ones (interactions with sites) to be interleaved in any order. This high degree of non-determinism may not always be desirable, as described in Section 4.2 of [20]. For example, in the expression $\text{DelayedN}() \mid M$, the call to M may be delayed, thus defeating the purpose of prioritizing the call to M . In order to rule out such undesirable be-

$$\begin{array}{c} \frac{u \text{ fresh}}{M(c) \xrightarrow{M(c,u)} ?u} \text{ (SITECALL)} \quad \frac{?u \xrightarrow{!c} !c}{} \text{ (SITERET)} \\ \frac{!c \xrightarrow{!c} 0}{} \text{ (PUB)} \quad \frac{E(Q) =_{def} f \in D}{E(P) \xrightarrow{!c} f\{P/Q\}} \text{ (DEF)} \\ \frac{f \xrightarrow{!c} f'}{f \mid g \xrightarrow{!c} f' \mid g} \text{ (SYM1)} \quad \frac{g \xrightarrow{!c} g'}{f \mid g \xrightarrow{!c} f \mid g'} \text{ (SYM2)} \\ \frac{f \xrightarrow{!c} f'}{f > x > g \xrightarrow{!c} (f' > x > g) \mid g\{c/x\}} \text{ (SEQ1V)} \quad \frac{f \xrightarrow{!c} f' \quad l \neq !c}{f > x > g \xrightarrow{!c} f' > x > g} \text{ (SEQ1N)} \\ \frac{f \xrightarrow{!c} f'}{g \text{ where } x : \in f \xrightarrow{!c} g\{c/x\}} \text{ (ASYM1V)} \quad \frac{f \xrightarrow{!c} f' \quad l \neq !c}{g \text{ where } x : \in f \xrightarrow{!c} g \text{ where } x : \in f'} \text{ (ASYM1N)} \\ \frac{g \xrightarrow{!c} g'}{g \text{ where } x : \in f \xrightarrow{!c} g' \text{ where } x : \in f} \text{ (ASYM2)} \end{array}$$

Figure 1. Asynchronous semantics of Orc

haviors, a *synchronous semantics* is proposed in [20] by placing further constraints on the application of SOS semantic rules of Figure 1. The synchronous semantics is arrived at by distinguishing between *internal* and *external* events, and splitting the SOS transition relation $\xrightarrow{\quad}$ into two subrelations \xrightarrow{R} , and \xrightarrow{A} , and characterizing set-theoretically, the complementary subsets of expressions (quiescent vs. non-quiescent) to which they are respectively applied. As we shall see in Section 4.3, in our rewriting semantics this set-theoretic splitting into \xrightarrow{R} , and \xrightarrow{A} is captured by two strategies *ex* and *in* combined in an overall strategy *sync*. However, in the above asynchronous semantics and its synchronous refinement, time is not explicitly modeled: it is only modeled implicitly by the fact that some external events may not yet be available and the expression becomes quiescent. We fully address this pending issue in the context of the real-time rewriting semantics of Orc in Section 5. Specifically, in Section 5.1 we capture the desired generalization with explicit real-time semantics of the *sync* strategy in two alternative ways: (i) by means of the *sync-timed* strategy; and (ii) by a rewriting semantics with additional equational conditions that requires no strategies.

4. Instantaneous Rewriting Orc Semantics

In this section, we explain in some detail the rewriting semantics of the asynchronous SOS definitions of Figure 1. Then, in Section 4.3 we characterize the synchronous semantics as the restriction on the asynchronous semantics imposed by a suitable rewriting strategy. We call both semantics *instantaneous*, in the sense that time elapse is not yet modeled (this is done in Section 5). However, even this instantaneous semantics has a rich computational granularity, because within a given time interval various external responses can be received from the environment, so that in the sense of [20] the evaluation of an expression may go through several quiescent states, followed by processing of new internal events after each external event reception. Due to space limitations we cannot give all details; they can be found in [2], including (in Appendix I there) the complete Maude specification². The complete set of rewrite rules in our specification is also given in Appendix A.

Before giving the rewriting logic semantic definitions, we describe the basic infrastructure that is needed to facilitate specification of the semantic rules. In addition to the declarations given in Table 3, We assume the following sorted (meta-)variable declarations throughout the rest of the paper, except where otherwise

²The Maude specification can also be found at <http://cs.uiuc.edu/homes/alturki/ppdp07>.

D	\in	$d_1; \dots; d_n$ (A list of declarations)	$r \in \text{Orc program}$	$::=$	$D; f$
E	\in	Expression Name	$d \in \text{Declaration}$	$::=$	$E(Q) =_{\text{def}} f$
x, z	\in	Variable	$f, g \in \text{Expression}$	$::=$	$\mathbf{0} \mid M(P) \mid E(P)$
M	\in	Site			$\mid f g$
c	\in	Constant			$\mid f > x > g$
h	\in	Handle			$\mid f \textbf{ where } x : \in g$
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)	$p \in \text{Actual Parameter}$	$::=$	$x \mid c \mid M$
			$q \in \text{Formal Parameter}$	$::=$	x

Table 3. Extended syntax of Orc

indicated.

$$\begin{array}{lll}
m, n \in \text{Nat} & C \in \text{ConstList} & r, r' \in \text{Record} \\
\rho \in \text{MsgPool} & \sigma \in \text{Context} & h \in \text{Handle} \\
\hat{c} \in \text{PreConst} & e \in \text{Event} & t \in \text{EventList} \\
c \in \text{Const} & &
\end{array}$$

4.1 Semantic Infrastructure

Events. As for any labeled transition system, labels in the semantic rules in Figure 1 represent *events* generated as a result of a configuration evolving into another. Lists of such events characterize *traces of actions* that a configuration may exhibit. The four possible event constructors are shown below.

$$\begin{array}{l}
\langle _ , _ \rangle : \text{SiteName} \times \text{ConstList} \times \text{Handle} \times \text{Nat} \rightarrow \text{Event} \\
_? _ : \text{Handle} \times \text{Const} \times \text{Nat} \rightarrow \text{Event} \\
!! _ : \text{Const} \times \text{Nat} \rightarrow \text{Event} \\
\tau : \rightarrow \text{Event}
\end{array}$$

The timed event $M\langle C, h \mid n \rangle$ represents a site call made to site M at time n with actual parameter list C . h is a fresh handle name that uniquely identifies this particular call. On the other hand, a site return with return value c occurring at time n and responding to the call whose handle is h is represented by the event $h?c \mid n$. The third event operator, $!!c \mid n$, denotes publishing a value c at time n , and, finally, τ is a non-timed event representing a silent transition, as usual.

Handles. A *handle* is a name that distinguishes a given site call from all other unfinished site calls, which are calls waiting for a response from the environment. Because handle names are simple identifiers and are invisible to the Orc programmer, we represent a handle as a term $h(n)$, with n a natural number of sort Nat .

$$h : \text{Nat} \rightarrow \text{Handle}$$

By the `SITECALL` rule of Figure 1, fresh handle names need to be generated. This is accomplished by maintaining in a configuration the next handle name to be used, which is updated appropriately as the configuration evolves.

Contexts. Since expressions may be abstracted with expression names, an environment needs to be maintained by the configuration to resolve references to such names. This is achieved by having a context structure in a configuration. A *Context* is a set of declarations formed with an associative and commutative multiset union operator $(_, _)$, with mt as its identity element, and the multiset elements are terms of sort Decl (which is a subsort of Context).

$$\begin{array}{l}
mt : \rightarrow \text{Context} \\
_, - : \text{Context} \times \text{Context} \rightarrow \text{Context}
\end{array}$$

Initially, a context is created out of the declaration list of an Orc program (see Table 4) so that the following conditions hold: (1) a later declaration in the list hides all previous declarations with the same expression name; and (2) all declarations in the resulting context are visible to each other. This implies that in a context,

an expression name has a unique defining declaration, and that (mutual) recursion is directly available.

Messages. Site calls and returns involve wide-area communications. To model such communications, we introduce a *message pool*, as a multiset of messages, into an Orc configuration. A *message* is a triple of the form $[M, C, h]$, where M is a site name to which the message is targeted, C is either a list of constants or a term of sort PreConst (more on this below), and finally h is a handle name identifying the call that caused this message. Since not all triples $[M, C, h]$ are valid messages, the kinds [15] $[\text{ConstList}]$ and $[\text{Msg}]$ are used instead of the sorts ConstList and Msg .

$$\langle _ , _ , _ \rangle : \text{SiteName} \times [\text{ConstList}] \times \text{Handle} \rightarrow [\text{Msg}]$$

Incoming messages to the configuration and outgoing messages to the environment share the same format. In a message $\gamma = [M, C, h]$, if M is the term *self* (representing a reference back to the configuration) and C is a term of sort PreConst (which subsumes the case where C is a constant value of sort Const , since Const is a subsort of PreConst), then γ is an incoming message and represents a (potential) response that is waiting in the message pool to be consumed by the configuration. On the other hand, if M is a site name other than *self* and C is a list of constants, then γ is an outgoing message destined for M , that was emitted into the pool as a result of executing a site call. Otherwise, γ does not represent a valid message. All this is specified compactly in membership equational logic using kind-level operators and (conditional) membership axioms to characterize valid messages of sort Msg .

$$\begin{array}{l}
[\text{self}, \hat{c}, h] : \text{Msg} \\
[M, C, h] : \text{Msg} \text{ if } M \neq \text{self}
\end{array}$$

Configurations. An Orc configuration constitutes a state of the system. A *configuration* consists of an Orc expression and a record.

$$\langle _ , _ \rangle : \text{Expr} \times \text{Record} \rightarrow \text{Conf}$$

A *record* is a set of fields (built with an associative-commutative set union operator $_ \mid _$), where each field represents a piece of information that we keep track of as the configuration evolves. In our case, five fields are maintained in a record. These are: the *trace* of events ($tr : t$), the *context* ($con : \sigma$), the *clock* ($clk : \text{clock}(n)$), the *pool of messages* ($msg : \rho$), and the *next available handle name* ($hdl : h$). As explained in Section 2.2, besides configurations $\langle P, R \rangle$, we also allow variants $\{P, R\}$ and $[P, R]$ to model single-step rewrites.

Having introduced the required infrastructure, we are now ready to discuss the rewriting semantics rules next.

4.2 Rewriting Semantics Rules

Site calls and returns. Site calls and site returns are specified using two rewrite rules. The first of these rules models a site call.

SITECALL:

$$\begin{aligned} & \{M(C), tr : t \mid msg : \rho \mid hdl : h(n) \mid clk : clock(m) \mid r\} \\ & \rightarrow [?h(n), tr : t . M\langle C, h(n) \mid m \rangle \mid msg : \rho \mid [M, C, h(n)] \mid \\ & \quad hdl : h(s(n)) \mid clk : clock(m) \mid r] \end{aligned}$$

The strictness of site calls is respected in the above rule by requiring the list of actual parameters to be a list of constant values (naturals, booleans, a signal, ... etc). When such a site call is encountered, the site call is replaced by the special expression ($?h(n)$), where $h(n)$ is the fresh handle name maintained in the configuration. At the same time, a message targeted to M is emitted into the message pool, a site call event is appended to the trace, and the handle counter is updated using the successor function s .

In the Orc SOS rules the environment is treated as a “black box”. This is reasonable, since responses from remote site calls are unpredictable. However, to obtain an executable Orc semantics that can be used as an interpreter, we somehow need to simulate environment responses. This is done as follows: once the message $[M, C, h(n)]$ is emitted into the message pool, it is converted into the message

$$[self, app(M, C, rand), h(n)],$$

which represents a *potential* response back to *self*. This message contains as its contents the operation *app* applied to three arguments. *app* is an operation of sort *PreConst* whose definition depends on the value of its arguments. It serves two purposes. First, it provides a uniform and abstract means by which the response of a particular site can be modularly defined. Second, it associates a pseudo-random delay, given by *rand* above, to responses of (external) sites, with the operational meaning that a well-formed response is not generated until the delay reaches the value zero. Once the delay is zero (and assuming the external site was known to the environment), the term $app(M, C, 0)$ is evaluated according to the value of M to a constant value (a ground term of sort *Const*). Only then, the response is ready to be consumed by the configuration, which is modeled by the site return rule below.

SITERET:

$$\begin{aligned} & \{?h, tr : t \mid msg : \rho \mid [self, c, h] \mid clk : clock(m) \mid r\} \\ & \rightarrow [!c, tr : t . h?c \mid m] \mid msg : \rho \mid clk : clock(m) \mid r] \end{aligned}$$

Besides consuming the message, the rule above replaces ($?h$) with the expression publishing the value obtained, and generates the appropriate event.

Publishing a value. The rule for publishing a value is quite straightforward. The expression is replaced by $\mathbf{0}$, and the appropriate event is generated.

$$\begin{aligned} \text{PUB} : & \{!c, tr : t \mid clk : clock(m) \mid r\} \\ & \rightarrow [\mathbf{0}, tr : t . (!c \mid m) \mid clk : clock(m) \mid r] \end{aligned}$$

Like a site call, publishing a value is strict, as it requires the variable to be substituted with its value before its publishing takes place.

Expression calls. Unlike site calls, expression calls are not strict. The actual parameter list of an expression call need not be all constants for the call to be evaluated.

$$\begin{aligned} \text{DEF} : & \{E(P), tr : t \mid con : \sigma, E(Q) =_{def} f \mid r\} \\ & \rightarrow [f\{P/Q\}, tr : t . \tau \mid con : \sigma, E(Q) =_{def} f \mid r] \end{aligned}$$

Using call-by-name semantics, the call is replaced with an instance of the body of the corresponding defining equation, where actuals are substituted for the formals one at a time³. Moreover, a τ

transition is recorded. Clearly, an expression call may entail an arbitrarily complex computation, which may evaluate to a (possibly empty) stream of values.

Sequential Composition. There are two cases, describing how two sequentially composed expressions, $f > x > g(x)$, may evolve. The first is when f publishes a value c while evolving to f' . In this case, a new instance of g having c substituted for x is created and is run in parallel with the (now evolved) composition $f' > x > g(x)$, while a τ event is generated. Thus, for each value c published by the evolution of f , a new instance $g\{c/x\}$ of $g(x)$ is created.

$$\begin{aligned} \text{SEQIV} : & \{f > x > g, tr : t \mid r\} \\ & \rightarrow [(f' > x > g) \mid g\{c/x\}, tr : t . \tau \mid r'] \\ & \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : (!c \mid m) \mid r'] \end{aligned}$$

The other case, where f evolves while generating an event other than publishing a value, is straightforward and is dealt with using three other sequential composition rewrite rules.

Symmetric parallel composition. The semantic rule for parallel composition is straightforward and resembles that of a process calculus. It merely stipulates that expressions running in parallel are allowed to evolve concurrently. Since the operator (\mid) is assumed associative and commutative, only one instance of the rule is required.

Asymmetric parallel composition. In an expression of the form $g(x)$ **where** $x : \in f$, the semantic rules allow g and f to evolve concurrently, unless f publishes a value. When f publishes a value c , the composition is replaced by $g\{c/x\}$. The rewrite rule that does just that is shown below.

ASYMIV:

$$\begin{aligned} & \{g \text{ where } x : \in f, tr : t \mid r\} \rightarrow [g\{c/x\}, tr : t . \tau \mid r'] \\ & \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : (!c \mid m) \mid r'] \end{aligned}$$

Of course, any subexpression of g that requires the value of x in order to make any progress would need to wait for f to publish its first value. The other cases for asymmetric parallel composition are similarly defined.

The key point about the above rewriting Orc semantics is that it *faithfully mirrors* the SOS Orc semantics from [20] given in Figure 1. This follows from three key observations: (i) the SOS semantics can first be put in MSOS format: this is a straightforward, mechanical transformation; (ii) the rewrite rules in the theory \mathcal{R}_{Orc} described above, and in Appendix A, are, except for the additional rules labeled COUNT, EVAL, and RAND which we have added for execution purposes, the exact translation of the MSOS Orc rules by the transformation from MSOS to rewriting logic summarized in Section 2.2 and described in full detail in [16]; and (iii) by Theorem 1 in [16], there is a *strong bisimulation* between the MSOS semantics of Orc and its corresponding rewriting logic semantics. Specifically, the corresponding rewriting logic semantics is obtained by removing from \mathcal{R}_{Orc} the rules labeled COUNT, EVAL, and RAND added for execution purposes. Appendix A lists all rules.

4.3 The Synchronous Execution Strategy

The rewrite theory described above does not enforce any execution strategy among instantaneous transitions of an Orc configuration. It reflects the exact behavior of the SOS semantics specification of Figure 1, which is in some sense too loose. In particular, site returns may take place in an expression while site calls that are

tuted with a variable, a constant, or a site name. Moreover, since we use the CINNI explicit substitution calculus [26], whenever renaming is needed to avoid free variable capture, CINNI automatically reflects it in the substitution term, keeping track of the right substitution (see [2, 1] for details).

³Substituting one variable at-a-time does not pose a problem as it is here equivalent to being done simultaneously. A variable can only be substi-

ready to be made are waiting. In what follows, we describe how, in agreement with the synchronous semantics of [20], internal actions (site calls, expression calls, and publishing of values) are given precedence over the external action of receiving responses from the environment using Maude’s strategy language.

First, the following “*in*” strategy is defined, where SEQ and ASYM respectively denote the set of labels of rewrite rules for sequential and asymmetric parallel compositions.

$$in := \text{SITECALL} \mid \text{PUB} \mid \text{DEF} \mid \\ \text{SYM}[in] \mid \text{SEQ}[in] \mid \text{ASYM}[in]$$

The strategy “*in*” applies one of the *internal action rules*. Note that the strategy expression has to be recursive, since we must make sure that only an internal action rule is applied while checking a condition of a conditional rule. Similarly, the following “*ex*” strategy, which applies the *site return rule*, either at the top or in conditions of the SYM, SEQ or ASYM rules, is defined.

$$ex := \text{SITERET} \mid \text{SYM}[ex] \mid \text{SEQ}[ex] \mid \text{ASYM}[ex]$$

Now, the complete strategy expression specifying the desired Orc execution behavior, can be given as

$$sync := \text{STEP}[in ? idle : ex]^+$$

where STEP is the label of the step rule described in Section 2.2, which represents a single step in the evolution of an Orc configuration. At each step, the substrategy *in ? idle : ex* controls how the condition of the step rule is checked. It tries (recursively) to match and apply an internal action. If it succeeds, the resulting configuration is returned and the condition is satisfied. In this case, the step is taken (at the top) with an internal action (this corresponds to a \hookrightarrow_A step in the sense of [20]). Otherwise, the external strategy is attempted on the original Orc configuration (corresponding to a \hookrightarrow_R step in the sense of [20]). If both substrategies fail, the condition is not satisfied and thus the step rule is not taken. In the next section, we will show how this strategy can be extended to support the timed semantics of Orc.

5. Timed Rewriting Orc Semantics

One important aspect of Orc that is outside the scope of the SOS semantic definitions of Figure 1 is that of time elapse. Several fundamental sites such as *atimer* and *rtimer* provide services whose meaning is dependent on time in a very precise and exact way, so this needs to be modeled. The point is that even though responses from *atimer* and *rtimer* are external events in the sense of [20], these are nevertheless *local* sites for each Orc program, which do not experience the unpredictable time delays and communication failures inherent in the computational model for the responses from *non-local* sites such as, say, CNN. Therefore, although no strong guarantees may be given about non-local site invocations, nevertheless, due to its real-time character, an Orc program may provide very strong guarantees for its behavior with respect to local site invocations. In this section we give a formal specification of such a real-time semantics. For this purpose, as usual for rewriting logic semantic definitions of real-time systems [23], we use a (discrete) time domain (maintained by the *clock(n)* field in a configuration), and a “tick” rewrite rule to advance time:

$$\text{TICK} : \{f, clk : clock(n) \mid r\} \rightarrow [f, clk : clock(s(n)) \mid \delta(r)]$$

where *s* is the successor function. The function δ propagates the effect of a clock tick down the record structure of a configuration. For instance, it updates time delays of messages in the message pool. By updating time delays, response messages from site calls become eventually available. It also updates contents of messages containing relative timing information, such as responses from the *rtimer(t)* site.

5.1 The Timed Execution Strategy

By dealing with time explicitly, we are adding another dimension along which Orc configurations could evolve. Care should be taken to avoid introducing behaviors that are uninteresting or undesirable. For instance, an Orc configuration that could take 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. Under this semantics, an Orc configuration can be seen to evolve along two axes in a two-dimensional plane. One axis is time, which is determined by discrete time clock ticks. The other axis encompasses all other computations of the system, which are the *instantaneous* transitions performed in a *synchronous* way. Instantaneous computations are given precedence over ‘tick’ computations, in the sense that the system is always allowed to evolve along the second ‘instantaneous’ axis as long as it can before the next tick happens. Once it reaches a state where it can no longer proceed along this instantaneous dimension, it takes a single step forward in time and then the process is again repeated in this fashion.

Despite its usefulness in eliminating some undesirable behaviors, the timed semantics sketched above has a limitation, as illustrated by the following example. Suppose that we have the declaration $E =_{def} let(0) > x > E()$. Now, using the above-mentioned semantics, an Orc configuration whose expression is $E()$ will prevent time from ever advancing. However, for what we call “*instantaneously terminating*” Orc programs, such as the *Metronome* program described in Section 3.2, where the expression evaluation always terminates within any single clock tick, this limitation is avoided. Therefore, in our semantics, we assume that such non-instantaneously terminating Orc programs are excluded.⁴

We describe below two approaches to specifying this timed strategy in rewriting logic: one using Maude’s strategy language, and the other purely equational and not requiring any strategies. Since an implementation of the subset of Maude’s strategy language that we use here is still under development as of this writing, the equational approach has the advantage that it is currently executable and, furthermore, can be subjected to formal analysis by model checking.

The Strategy Language Approach

To achieve the behavior described above, we can easily extend the strategy expression *sync-instant* given in Section 4.3 for instantaneous transitions, so that the tick rule is taken into account. The new strategy is

$$sync-timed := \text{STEP}[in ? idle : (ex ? idle : \text{TICK})]^+$$

In this strategy, internal transitions are given precedence over external site response transitions, which are, in turn, given precedence over the clock tick transition.

In the presence of delays, a simpler strategy having a similar effect to the strategy above may be specified. More specifically, assuming non-zero delays, responses from external sites are not consumed by an Orc expression before at least one clock tick takes place, and thereby having the effect of giving precedence to internal actions over the site return action. This simpler strategy can be

⁴ We do not address the pragmatic issue of instantaneously terminating Orc programs doing so within reasonable bounds. Having some bounds (for example in number of rewrites needed) for their instantaneous termination is of course important for the granularity of clock ticks that are then feasible in practice.

specified by the following expression:

$$timed := STEP[eager ? idle : TICK]^+$$

where *eager* is defined as

$$eager := SITECALL | PUB | DEF | SITERET | SYM | SEQ | ASYM$$

with SEQ and ASYM standing, respectively, for the labels of the sequential and asymmetric parallel composition rules, as before. Note that the strategy expression *timed* need not be recursive, since the tick rule cannot match any of the rewrite conditions of the instantaneous rules.

An Equational Approach

The effect of giving to the tick rule above the least priority possible can instead be achieved by making the rule conditional to an *eagerEnabled* predicate as follows:

$$TICK : \{f, clk : clock(n) \mid r\} \rightarrow [f, clk : clock(s(n)) \mid \delta(r)] \\ \text{if } eagerEnabled(\{f, clk : clock(n) \mid r\}) \neq true$$

The predicate *eagerEnabled* is defined using a technique similar to the one proposed in [24]. In this method, a predicate named *eagerEnabled* on configurations is declared, which, given a configuration \mathcal{C} , should evaluate to *true* if and only if there exists an eager (that is, instantaneous) rule using which \mathcal{C} could rewrite to some other configuration. The approach of [24], however, is not directly applicable to our setting, because it assumes that rules have no rewrites in their conditions. Here, we introduce a variant of that approach that overcomes this limitation by taking advantage of some of the properties of our specifications. We first declare the predicate as a partial function as follows,

$$eagerEnabled : Conf \rightarrow [Bool] \ [frozen]$$

where the predicate is declared as a *frozen* operator to avoid useless rewrites in the configuration. Then, for each eager (that is, non-tick) rule $r : \{E, R\} \rightarrow [E', R']$ if $C \wedge \bigwedge_{i=1}^n \{E_i, R_i\} \rightarrow [E'_i, R'_i]$ in our rewrite theory, with C a possibly empty conjunction of equational conditions (memberships and/or equations) and $n \geq 0$, we introduce an equation

$$eagerEnabled(\{E, R\}) = true \text{ if } C \wedge \bigwedge_{i=1}^n eagerEnabled(\{E_i, R_i\})$$

Intuitively, this states that a configuration is *eager* if there exists a rewrite rule that matches the configuration and is such that: (i) its equational conditions are satisfied under this matching, and (ii) the controls of its rewrite conditions are configurations that are themselves eager. Finally, the application of the tick rule is subjected to the condition that the configuration is not eager. The reader is referred to [2] for a detailed description and a proof of correctness of a general construction of the *eagerEnabled*, not just for the Orc case, but for a large class of rewrite theories encompassing in practice many rewrite theories modeling small-step SOS semantics.

Note that this specification using the *eagerEnabled* predicate is equivalent to the strategy '*timed*' given above in the strategy language. The same construction can be used to equationally specify a rewrite theory whose behavior is equivalent to the '*sync-timed*' strategy, by using (in addition to the *eagerEnabled* predicate) another predicate, called *intAction*, for which the site return rule is the "lazy" rule and the internal action rules are the "eager" rules.

5.2 Site Definitions

In order to be able to experiment with the above real-time semantics of the Orc language and execute and model check programs, sites,

especially fundamental ones, need to be specified. One important design goal of this work was to keep definitions of sites separate from Orc definitions, as they are supposed to be. This has been achieved, in part, by defining the abstract application function *app*. Then, separate modules that define sites can be declared and can be used to give concrete definitions of the *app* function for each site of interest. An example Maude specification of a simple site module is that of the *if(b)* site shown below, which publishes a signal when called with the value `tr(true)` (representing the truth value *true*).

```
mod IF-SITE is
  inc ORC-SEMANTICS .
  op if : -> SiteName [ctor] .
  eq app(if, tr(true), 0) = sig .
endm
```

First, the site name is syntactically introduced, and then the semantics of *app* is defined for it, with *sig* being a constant representing a *signal*. Note that `app(if, tr(false), 0)` is a *Pre-Const* term that does not reduce to any ground *Const* term, modeling a site not responding.

To add some basic computational power to our Orc specification, we also define sites performing basic arithmetic functions, binary relations, and binary logical operations.⁵

6. Formal Analysis of Orc Programs

In this section we illustrate how the real-time operational semantics we have developed can be used to experiment with Orc programs, explore traces of computations, and verify properties about Orc programs. We first give the syntax of Orc specified in Maude⁶ in Table 4. Since Maude supports mixfix user-definable syntax, the syntax in the Maude specification is a readable, typewriter version of the original Orc syntax. In Maude, argument positions are indicated by underscores. For example, the *where* composition operator is declared with syntax `_where_ : in_` approximating the original syntax `_where_ : ∈_`.

Based on the algebraic properties of the Orc language constructs [20], the sequential and asymmetric parallel composition operators are declared right associative, while the symmetric parallel operator is fully associative, commutative, and has the identity *zero*. Furthermore, the left annihilator axiom of sequential composition is specified with an equation (using the `eq` keyword), as follows.

$$eq \text{ zero } > X > E = \text{ zero } .$$

In addition to the operators of Table 4, a few syntactic sugar operators are defined so that the empty lists `nilA` and `nilF` need not be given (e.g. a call with no actual parameters can be written as `S()` instead of `S(nilA)`).

⁵Responses from the sites *clock*, *signal*, *atimer(t)*, and *rtimer(t)* are not subjected to delays to preserve their meaning. *let* is also not subjected to delays as it is assumed to be local to the expression being evaluated.

⁶Note that in Maude each mixfix syntax declaration starts with an `op` (for "operator") followed by the mixfix syntax declaration itself, followed by ":", followed by the sorts (corresponding to nonterminals in a CF grammar) of the arguments, followed by ">" followed by the sort of expressions with that syntax. Precedence information can be added with the `prec` attribute, and left- and right-associativity information with the `gather` attribute. Furthermore, a binary syntax construct can be declared with *semantic* axioms such as associativity (`assoc`), commutativity (`comm`), and identity (`id`). In particular, associativity makes use of parentheses unnecessary, and commutativity makes the order of arguments immaterial (See Section 3 of the Maude manual [8]).

```

fmod ORC-SYNTAX is
  op _;_ : DeclList Expr -> Prog [prec 50] .
  op nilD : -> DeclList .
  op _;_ : DeclList DeclList -> DeclList
  op _:=_ : ExprName FParamList Expr -> Decl [prec 30] .
  op zero : -> Expr .
  op _(_) : SiteName AParamList -> Expr [prec 10] .
  op _(_) : ExprName AParamList -> Expr [prec 10] .
  op !_ : IVar -> Expr [prec 5] .
  op !_ : Const -> Expr [prec 5] .
  op _>_ : Expr Var Expr -> Expr
    [prec 15 gather (e & E)] .
  op _|_ : Expr Expr -> Expr
    [assoc comm id: zero prec 20] .
  op _where_:in_ : Expr Var Expr -> Expr
    [prec 25 gather (E & e)] .
  op ?_ : Handle -> Expr [prec 1] .
endfmod

```

Table 4. An excerpt from the functional module ORC-SYNTAX which specifies the extended syntax of Orc in Maude.

Using these syntactic specifications, the program PRIORITY given in Section 3.2, for instance, is represented as a term of the theory ORC-SYNTAX as follows⁷:

```

'eDelayedN := rtimer(1) > 'vz >
  (let('vu{0}) where 'vu :in 'sN()) ;
let('vx{0}) where 'vx :in 'sM() | 'eDelayedN()

```

Before we get to the examples, a vital observation is in order. Recall that time is kept track of using the `clock(n)` operator as part of a configuration. This may directly cause the number of states reachable from a given configuration to be infinite. Moreover, the state space of the pseudo-counter used to generate random delays is the set of natural numbers and, thus, causes the state space of an Orc configuration to grow indefinitely, even for configurations that are originally finite-state, which may severely limit our ability to analyze programs in the language. To resolve these issues, we set these parameters so that the number of clock ticks is limited to the first ten ticks, and the counter to the first five natural numbers, implying that at most five pseudo-random numbers are generated. These two parameters can be easily changed to better suit the example at hand by appropriately adjusting the following two equations,

```
eq clock(9) = halt . and eq s_~5(counter) = counter .
```

Consider the program TIMEOUT given in Section 3.2. We can simulate a run of the program using Maude's `rew` command, assuming that M is a site that returns the value 1 (the operator `[P]` constructs an initial configuration given an Orc program P).

```

Maude> rew ['eF 'vt := let('vz{0}) where 'vz :in ('sM()
  | rtimer('vt{0}) > 'vx > ! 0) ; 'eF(3)] .
rewrites: 2004 in 60ms cpu (85ms real)
(33400 rewrites/second)
result Conf: < zero,(tr : tau . ('sM < nilA,h(0) | 0 >)
. (rtimer < 3,h(1) | 0 >) . (h(1) ? sig | 3) . tau .
tau . (let < 0,h(2) | 3 >) . (h(2) ? 0 | 3) . !! 0 | 3)
| (con : 'eF 'vt := let('vz{0}) where 'vz :in 'sM(nilA)
| rtimer('vt{0}) > 'vx > ! 0) | (clk : halt) | (msg :
[self,1,h(0)]) | hdl : h(3) >

```

The execution trace, given by the field indexed by `tr`, shows the events that took place to reach the resulting configuration. The trace

⁷Note that names are specified using quoted identifiers, with the different classes of names distinguished by the first letter following the quote as follows: variable names start with `v`, site names with `s`, and expression names with `e`. For example, `'vx` stands for the variable x , whereas `'sM` denotes the site M .

shows that the call to M has timed out, and thus the value 0 was published (at clock tick 3). By increasing the timeout to, say 6, we get the following run.

```

Maude> rew ['eF 'vt := let('vz{0}) where 'vz :in ('sM()
  | rtimer('vt{0}) > 'vx > ! 0) ; 'eF(6)] .
rewrites: 1980 in 70ms cpu (102ms real)
(28285 rewrites/second)
result Conf: < zero,(tr : tau . ('sM < nilA,h(0) | 0 >)
. (rtimer < 6,h(1) | 0 >) . (h(0) ? 1 | 5) . tau .
(let < 1,h(2) | 5 >) . (h(2) ? 1 | 5) . !! 1 | 5) |
(con : 'eF 'vt := let('vz{0}) where 'vz :in 'sM(nilA)
| rtimer('vt{0}) > 'vx > ! 0) | (clk : halt) | (msg :
[self,sig,h(1)]) | hdl : h(3) >

```

In this run, the response from M (the value 1) is the value published by the expression, since the response was delayed by 5 time units, which is less than the timeout.

We can also verify some safety properties of Orc programs using the breadth-first search command of Maude. As an example, consider the program TIMED-MCALL. We can verify that the property that no two calls to M occur at the same time is satisfied in any state that the program could evolve to. This can be achieved by issuing the following search command, for which no solution exists, as expected (the arrow `=>*` stands for zero, one or more rewrites starting from the given configuration).

```

Maude> search [nilD ; 'sM() | rtimer(1) > 'vx > 'sM() |
rtimer(2) > 'vx > 'sM() | rtimer(3) > 'vx > 'sM()]
=>* < E:Expr , (tr : e:EventList .
('sM < nilA, H:Handle | N:Nat >) .
e':EventList . ('sM < nilA, H':Handle | N:Nat >) .
e'':EventList) | R:Record > .

```

```

No solution.
states: 255017 rewrites: 14319638 in 840540ms cpu
(896223ms real) (17036 rewrites/second)

```

Using Maude's LTL model checking capabilities, one can verify more complex safety and liveness properties of finite-state systems. We use the well-known Dining Philosophers problem (DF), of which a specification in Orc is given in [20], to illustrate some of these capabilities⁸.

$$P_i := \text{fork}[i](\text{get}) \gg \text{fork}[i'](\text{get}) \gg \text{eat}[i]() \gg \text{fork}[i](\text{put}) \gg \text{fork}[i'](\text{put}) \gg P_i()$$

with $i' = i + 1 \bmod n$. The full specification of DF in Maude can be found in [2]. An operator `df(n)` is used to construct the above solution of DF with n philosophers.

A fundamental property of DF is *relative exclusion*, which asserts that no two adjacent philosophers may eat at the same time. This property is specified using the `rel-excl(n)` operator defined below, with n the number of philosophers,

```

eq rel-excl(n) = [] re(n - 1, n) .
eq re(s(m), n) = ~(eats(s(m)) /\ eats((s(m) + 1) rem n))
  /\ re(m,n) .
eq re(0, n) = ~(eats(0) /\ eats(1)) .

```

where the parameterized *eats* predicate is defined as follows.

```

eq < (eat() > Y > E) > Y' > phil[i]() | E', R >
  |= eats(i) = true .

```

The predicate *eats(i)* is true in any state in which the i th philosopher is currently eating, i.e. P_i is ready to call the *eat* site. Now we show

⁸For this example, we assume no delays and limit the clock to a single clock tick, since timing and delays are not relevant for this example and, thus, no interesting behavior is lost under these assumptions. Note that \gg is the special case of sequential composition in which no value is passed.

that the relative exclusion property is satisfied by $df(3)$ by issuing the following LTL model checking command in Maude.

```
Maude> red modelCheck( { df(3) } , rel-excl(3) ) .
rewrites: 1511157 in 33010ms cpu (34527ms real)
(45778 rewrites/second)
result Bool: true
```

However, the program $df(3)$ is not deadlock-free. This is because, for example, all philosophers may choose to pick their right forks first at the same time, in which case, they will all be waiting indefinitely for their left forks. To model-check this property we axiomatize it in our theory as follows. We first note that a configuration is *deadlocked* if it reaches a state where no transition (other than the one advancing the clock) can be taken. Therefore, by the definition of the *eagerEnabled* predicate, a configuration C being deadlocked coincides with the *eagerEnabled(C)* predicate not being true. Therefore, we define the *enabled* predicate accordingly.

```
ceq < E , R > |= enabled = true
if eagerEnabled({ E , R } ) .
```

Now, using this predicate and the operator `no-deadlock`, which is simply the LTL formula $[\] \text{ enabled}$, we can use the model checker to obtain a run of the configuration yielding a deadlock (the output of the run showing the counterexample is somewhat long and is mostly omitted here).

```
Maude> red modelCheck( { df(3) } , no-deadlock ) .
rewrites: 136741 in 2610ms cpu (2727ms real)
(52391 rewrites/second)
result ModelCheckResult: counterexample({< phil[0](nilA)
| phil[1](nilA) | phil[2](nilA), (tr : (nil).EventList)
| (con : (phil[0] nilF := fork[0](get) > 'vt > fork[1]
...
get, 0), h(0)) | (hdl : h(0) # h(1) # h(2)) | t-forks
: fork[0] . fork[1] . fork[2] >, deadlock})
```

One solution to the deadlock problem is to impose a restriction on the order in which the forks are picked up as follows. When the first philosopher is hungry, he picks up his left fork first and then his right fork. All other philosophers pick their right forks first. The operator $df-df(n)$ reflects this change to the DF specification given above. The new specification is verified deadlock-free by the model checker.

```
Maude> red modelCheck( { df-df(3) } , no-deadlock ) .
rewrites: 1547969 in 33660ms cpu (34969ms real)
(45988 rewrites/second)
result Bool: true
```

In [2], the SOS-based semantics given here is compared to a more efficient, semantically equivalent rewriting semantics. The more efficient semantics is obtained by following a reduction semantics style in which the number of conditional rewrite rules is minimized, and the rewrite conditions are eliminated. The reader is referred to [2] for a detailed discussion of the reduction rewriting semantics of Orc.

7. Concluding Remarks

We have given a formal real-time operational semantics for Orc programs based on rewriting logic. Both time elapse, and the different execution priorities given to internal and external events by an Orc program are faithfully modeled. Furthermore, the Maude specification of this operational semantics definition provides both an Orc interpreter and an Orc LTL model checker. Our approach has some similarities with the various SOS semantics that have been given for different timed process calculi, such as ATP [22] and TLP [11], and real-time extensions to various process calculi,

such as extensions of ACP [4, 3], CCS [6], and CSP [25]. However, the alternative possibility of giving a faithful Orc semantics by translating Orc into some of these timed process calculi seems highly nontrivial.

The SOS-based rewriting semantics given here paves the way for a more efficient rewriting specification in which deterministic Orc features are modeled with equations, and only the non-deterministic features of Orc are modeled using rewrite rules, and where, furthermore, rules are made local (not restricted to work only on configurations) and unconditional as much as possible. Some progress has already been made in this direction. We intend to continue this work with the goal of arriving at a physically distributed deployment of Orc using the socket programming capabilities of Maude [7]. Such a distributed deployment would provide a rich formal environment for the experimentation, analysis and verification of Orc programs, which could then be extended towards a full-fledged, rewriting-based Orc implementation with associated analysis and verification tools.

Acknowledgments

Supported in part by ONR Grant N00014-02-1-0715.

References

- [1] M. AlTurki. A rewriting logic approach to the semantics of Orc. Master's thesis, University of Illinois at Urbana-Champaign, USA, December 2005.
- [2] M. AlTurki and J. Meseguer. Rewriting logic semantics of Orc. Technical report, Department of Computer Science, UIUC, Urbana-Champaign, USA, 2007.
- [3] J. Baeten and C. Middelburg. Process algebra with timing: Real time and discrete time, 2000.
- [4] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Asp. Comput.*, 3(2):142–188, 1991.
- [5] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1-3):386–414, 2006.
- [6] L. 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.
- [7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*. To be published by Springer, 2007.
- [8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.3). January 2007. <http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf>.
- [9] W. R. Cook and J. Misra. A structured orchestration language. July 2005. <http://www.cs.utexas.edu/users/wcook/Drafts/OrcCookMisra05.pdf>.
- [10] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In *Proceedings of STRATEGIES'06*, August 2006.
- [11] M. Hennessy and T. Regan. A process algebra for timed systems. *Inf. Comput.*, 117(2):221–239, 1995.
- [12] T. Hoare, G. Menzel, and J. Misra. A tree semantics of an orchestration language. August 2004. Also available at <http://www.cs.utexas.edu/users/psp/Semantics.Orc.pdf>.
- [13] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.
- [14] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

- [15] J. 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.
- [16] J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. In *AMAST*, pages 364–378, 2004.
- [17] 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.
- [18] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 2007. To appear (preliminary version in Proc. SOS'05), <http://fs1.cs.uiuc.edu/pubs/meseguer-rosu-2006-tcs.pdf>.
- [19] J. Misra. Computation orchestration: A basis for wide-area computing. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktobendorf, Germany, 2004.
- [20] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006.
- [21] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.
- [22] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.
- [23] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
- [24] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. *Electr. Notes Theor. Comput. Sci.*, 117:285–314, 2005.
- [25] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. 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.
- [26] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ -, ζ - and π -calculi. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.

A. The Instantaneous Rewriting Semantics Rules of Orc in Maude

STEP:

$$\langle f, r \rangle \rightarrow \langle f', r' \rangle \text{ if } \{f, r\} \rightarrow [f', r']$$

SITECALL:

$$\begin{aligned} & \{M(C), tr : t \mid msg : \rho \mid hdl : h(n) \mid clk : clock(m) \mid r\} \\ & \rightarrow [?h(n), tr : t . M\langle C, h(n) \mid m \rangle \mid msg : \rho \mid [M, C, h(n)] \mid \\ & \quad hdl : h(s(n)) \mid clk : clock(m) \mid r] \end{aligned}$$

SITERET:

$$\begin{aligned} & \{?h, tr : t \mid msg : \rho \mid [self, c, h] \mid clk : clock(m) \mid r\} \\ & \rightarrow [!c, tr : t . h?c \mid m] \mid msg : \rho \mid clk : clock(m) \mid r \end{aligned}$$

PUB:

$$\begin{aligned} & \{!c, tr : t \mid clk : clock(m) \mid r\} \\ & \rightarrow [0, tr : t . (!!c \mid m) \mid clk : clock(m) \mid r] \end{aligned}$$

DEF:

$$\begin{aligned} & \{E(P), tr : t \mid con : \sigma, E(Q) =_{def} f \mid r\} \\ & \rightarrow [f\{P/Q\}, tr : t . \tau \mid con : \sigma, E(Q) =_{def} f \mid r'] \end{aligned}$$

SYM:

$$\begin{aligned} & \{f \mid g, tr : t \mid r\} \rightarrow [f' \mid g, tr : t . L \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : L \mid r'] \end{aligned}$$

SEQ1V:

$$\begin{aligned} & \{f > x > g, tr : t \mid r\} \\ & \rightarrow [(f' > x > g) \mid g\{c/x\}, tr : t . \tau \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : (!!c \mid m) \mid r'] \end{aligned}$$

SEQ1N1:

$$\begin{aligned} & \{f > x > g, tr : t \mid r\} \rightarrow [f' > x > g, tr : t . \tau \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : \tau \mid r'] \end{aligned}$$

SEQ1N2:

$$\begin{aligned} & \{f > x > g, tr : t \mid r\} \rightarrow [f' > x > g, tr : t . h?c \mid m \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : h?c \mid m \mid r'] \end{aligned}$$

SEQ1N3:

$$\begin{aligned} & \{f > x > g, tr : t \mid r\} \\ & \rightarrow [f' > x > g, tr : t . M\langle C, h \mid m \rangle \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : M\langle C, h \mid m \rangle \mid r'] \end{aligned}$$

ASYM1V:

$$\begin{aligned} & \{g \textbf{ where } x : \in f, tr : t \mid r\} \rightarrow [g\{c/x\}, tr : t . \tau \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : (!!c \mid m) \mid r'] \end{aligned}$$

ASYM1N1:

$$\begin{aligned} & \{g \textbf{ where } x : \in f, tr : t \mid r\} \\ & \rightarrow [g \textbf{ where } x : \in f', tr : t . \tau \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : \tau \mid r'] \end{aligned}$$

ASYM1N2:

$$\begin{aligned} & \{g \textbf{ where } x : \in f, tr : t \mid r\} \\ & \rightarrow [g \textbf{ where } x : \in f', tr : t . h?c \mid m \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : h?c \mid m \mid r'] \end{aligned}$$

ASYM1N3:

$$\begin{aligned} & \{g \textbf{ where } x : \in f, tr : t \mid r\} \\ & \rightarrow [g \textbf{ where } x : \in f', tr : t . M\langle C, h \mid m \rangle \mid r'] \\ & \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : M\langle C, h \mid m \rangle \mid r'] \end{aligned}$$

ASYM2:

$$\begin{aligned} & \{g \textbf{ where } x : \in f, tr : t \mid r\} \\ & \rightarrow [g' \textbf{ where } x : \in f, tr : t . L \mid r'] \\ & \text{if } \{g, tr : nil \mid r\} \rightarrow [g', tr : L \mid r'] \end{aligned}$$

RAND:

$$rand \rightarrow floor((random(counter)/4294967296) \times 10)$$

$$COUNT : counter \rightarrow s(counter) \quad EVAL : counter \rightarrow 0$$