Rewriting Logic Semantics of Orc

Musab AlTurki and José Meseguer
{alturki,meseguer}@uiuc.edu
University of Illinois at Urbana Champaign

November 20, 2007

Abstract

Orc is a language for *orchestration* of web services developed by J. Misra that offers simple, yet powerful and elegant, constructs to succinctly program sophisticated web orchestration 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 report, we first propose a real-time operational Orc semantics, that captures the informal operational semantics given in [26]. This operational semantics is given as a rewrite theory $\mathcal{R}_{Orc}^{sos}$ in which the elapse of time is explicitly modeled. This is followed by presenting a much more efficient *reduction semantics* of Orc, which is provably equivalent to the SOS semantics. A detailed proof of strong bisimilarity of the two semantic specifications is then given. In both theories, the priorities between internal and external events and the *time-synchronous* execution strategy used are modeled in two alternative ways: (i) by a rewrite strategy; and (ii) by adding extra equational conditions to the semantic rules. We show experiments demonstrating the much better performance of the reduction semantics when compared to the SOS semantics.

We view this reduction semantics as a key intermediate stage towards a future, provably correct distributed implementation of Orc. We describe a distributed, object-based view of the Orc model and its specification. Using the Maude rewriting logic language, we also illustrate how the distributed semantics can be used to endow Orc with useful formal analysis capabilities, including an LTL model checker and search for violations of invariants. We illustrate these formal analysis features by means of two applications: an online auction system and a meeting scheduler, both of which are modeled as distributed systems of actors that perform Orc computations.

Contents

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 [25]. 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 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 [26, 27, 17]. 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 this report, we use rewriting logic to capture the intended real-time semantics of Orc. We first present an SOS-like rewriting semantics that is directly based on the SOS semantics of [27]. Then, a much more concurrent *reduction semantics* of Orc is presented. This reduction semantics takes full advantage of rewriting logic's concurrent semantics and fully exploits rewriting logic's crucial distinction between equations and rewrite rules. We show how the real-time synchronous semantics of Orc can be faithfully captured in the reduction semantics and establish the semantic equivalence between the reduction semantics and the SOS-like semantics. We also provide experimental evidence for the claim that the reduction semantics is much more efficient than the SOS-like semantics.

The reduction semantics is a crucial step towards the refinement of Orc specifications into a distributed implementation. Indeed, 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, we propose a distributed specification that encapsulates 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 [20]. 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 two applications: an online Orc auction system and a meeting scheduler, both of which are realized as distributed collections of Orc expression and site objects.

2 Related Work

Several Orc semantics have already been given. A precise but informal operational semantics for Orc was given by Misra in [26]; 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 [27]; 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 [27]. Different denotational semantics of Orc for that are well-suited for reasoning about identities and algebraic laws in the language, rather than for describing the operational behavior of Orc programs, were also given [16, 17]. Moreover, a denotational semantics was given in [32], which used labeled event structures to analyze dependancies in program execution. Furthermore, Bruni, Melgratti, and Tuosto [5] gave encodings of Orc in Petri nets and the join calculus that reveal some of the subtleties of the semantics of the language. Most recently, Ian Wehrman et al. [35] proposed a relative-time operational semantics of Orc by extending the asynchronous SOS relation of [27] 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 [29] and TLP [15], and real-time extensions to various process calculi, such as extensions of ACP [2, 1], CCS [7], and CSP [33].

3 Preliminaries

3.1 Rewriting Logic as a Semantic Framework

Rewriting logic [19] 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 [18, 24] and references there), and to model real-time systems [30]. Three important aspects of rewriting logic have stimulated interest in using it for specifying languages:

1. The generality and flexibility of the rewriting semantic framework makes it suitable for specifying both the deterministic features of a language using equations, and the non-deterministic ones with rewrite rules, within a uniform model. This also provides a convenient way to control the level of abstraction desired in a specification.

2. With the availability of high-performance rewriting logic implementations, such as Maude [9], rewriting semantics specifications are immediately executable, providing an interpreter and an LTL model checker for the language specified, essentially for free.

3. The generic and efficient formal analysis tools developed for rewriting logic specifications are readily available for analyzing properties based on their rewriting semantics. These tools include, among others, a semi-decision procedure for checking violations of invariants by breadth-first search, an LTL model checker for finite state systems, an inductive theorem prover, confluence and coherence checkers, and a termination tool.

A rewrite theory is a formal description of a concurrent system, including its static state structure and its dynamic behavior. Since a rewrite theory has an underlying equational theory in some equational logic, we briefly describe specifications in a very general form of equational logic, namely Membership Equational Logic theories. This is followed by a general description of rewrite theories and a quick overview of the Maude system.

Membership Equational Logic

A Membership Equational Logic (MEL) *theory* [21] is a pair (Σ, E) with Σ a MEL signature and E a set of MEL Σ-sentences. A MEL *signature* is a triple (K, Σ, S) , where K is a set of kinds, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ is a $K^* \times K$-kinded family of sets of operators, and $S = \{S_k\}_{k\in K}$ is a K-kinded family of disjoint sets of sorts. Given a signature Σ, a Σ-algebra A is a structure that assigns a set A_k to each kind K, a function $f_A : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ to each operation symbol $f \in \Sigma_{k_1 \cdots k_n, k}$, and a subset $A_s \subseteq A_k$ to each sort $s \in S_k$. Two fundamental Σ-algebras are the Σ-term algebra of ground terms denoted by T_Σ, and the Σ-algebra of terms over a K-kinded set of variables $X = \{x_1 : k_1, \cdots, x_n : k_n\}$, denoted by $T_\Sigma(X)$. The kind of a sort s is denoted by $[s]$.

A MEL Σ-*sentence* is then a universally-quantified, conditional formula $(\forall X)\ \phi$ if $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$, with ϕ an atomic formula being either a Σ-equation of the form $t = t'$ (with t and t' of the same kind), or a Σ-membership of the form $t : s$ (with t of kind $[s]$), where all terms are terms in $T_\Sigma(X)$, and s and s_j are sorts in S. If $s_1, s_2 \in S$, the notation $s_1 < s_2$ declares s_1 a *subsort* of s_2, which abbreviates the Σ-sentence $(\forall x : k)\ x : s_2$ if $x : s_1$. Another notational convenience is that an operator can be declared at the sort level $f : s_1 \cdots s_n \to s$, which corresponds to declaring it at the kind level and adding the sentence $(\forall x_1 : k_1 \cdots x_n : k_n)\ f(x_1 \cdots x_n) : s$ if $\bigwedge_{1\leq i\leq n} x_i : s_i$. The Σ-membership $w_j : s_j$ asserts that w_j, which is of kind $[s_j]$, has sort s_j, representing a well-defined term w_j. This is in contrast to a term having a kind but not belonging to any sort, which is considered to be an undefined or error term. The introduction of kinds and sorts in MEL provides a natural and convenient way to model partial functions [21].

The *models* of a MEL theory (Σ, E) are (Σ, E)-algebras, that is, Σ-algebras that satisfy the equations in E. Of such models, the *initial algebra* $T_{\Sigma/E}$, whose elements are E-equivalence classes of Σ-terms modulo the provable equalities, and whose memberships are those provable in the logic, gives a denotational semantics to the theory (Σ, E). However, the operational semantics of the theory is given by equational simplification using the equations in E as rewrite rules.

For a detailed treatment of MEL, its models and deduction system, the reader is referred to [21, 3].

Rewriting Logic

In its most general form, a *rewrite theory* is a 4-tuple $\mathcal{R} = (\Sigma, E, R, \phi)$ with:

- (Σ, E) a MEL theory,
- R a set of universally quantified labeled *conditional rewrite rules* of the form:

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

  where terms appearing in an equation or a rewrite are of the same kind, X contains all the variables appearing in all terms, and for all j, the kind of r_j is $[s_j]$, and

- $\phi : \Sigma \to \mathcal{P}(\mathbb{N})$ 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 [19]. [6] gives a precise account of the most general form of rewrite theories and their models.

3.2 MSOS to Rewriting Logic Transformation

Modular structural operational semantics (MSOS) [28] 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 [23] 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

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

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.

$$\text{Step} : \langle P, R\rangle \to \langle P', R'\rangle \text{ if } \{P, R\} \to [P', R']$$

The reader is referred to [23] for a more detailed discussion of the methodology and a proof of its semantics-preserving correctness.

3.3 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.

As a simple example, the following functional module specifies multisets of natural numbers along with a (partial) function that computes the smallest element of a given multiset of elements.

```
fmod NAT-MSET is
  protecting NAT .
  sort NatMSet .
  subsort Nat < NatMSet .
  op mt : -> NatMSet [ctor] .
  op __ : NatMSet NatMSet -> NatMSet [ctor assoc comm id: mt] .
  op min : NatMSet -> [Nat] .
  vars N M : Nat . var  S : NatMSet .
  eq  min(N N S) = min(N S) .
  ceq min(N M S) = min(N S) if N < M .
  eq  min(N) = N .
endfm
```

The module `NAT-MSET` first includes Maude's built-in `NAT` module while *protecting* it from adding *junk* or introducing *confusion*. The other types of inclusion are the `extending` and the `including` module inclusions which are more relaxed than the protecting inclusion (see [8] and [9] for details). The sort `NatMSet` is then declared using the keyword `sort`, whereas subsorting information is given using the `subsort` keyword. Each syntax declaration starts with an `op` (for "operator"), followed by the syntax declaration itself, followed by "`:`", followed by the sorts or the kinds (corresponding to nonterminals in a CF grammar) of the arguments, followed by "`->`", followed by the sort or kind of expressions with that syntax. The `ctor` attributes declares the corresponding operator as a *constructor* operator, as opposed to *defined* operators. In this case, `mt` and the juxtaposition operator `__` are constructors for the sort `NatMSet`, while the function `min` is a defined operator. 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. Other attributes of operators can be given, such as precedence information using the `prec` attribute, and left- and right-associativity information with the `gather` attribute. Unconditional and conditional equations are introduced with the `eq` and `ceq` keywords, memberships with `mb` and `cmb`, and rewrite rules (in system modules) with `rl` and `crl`.

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 9. For a complete description of these features and tools, and various other tools provided by Maude, the reader is referred to the Maude book [8] and the Maude manual [9].

3.4 The Maude Strategy Language

Maude's strategy language [12] 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

L	$\in$	Labels
R	$\in$	Strategy names

$B \in$ Basic Strategy	$::=$	idle $\mid$ fail $\mid$ L $\mid$ $L[SL]$
$S, S_1, S_2 \in$ Strategy	$::=$	$R \mid B \mid S_1; S_2 \mid S_1 \vert S_2$
		$\mid S^* \mid S^+ \mid S?\ S_1 : S_2$
$SL \in$ 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

terms that can be obtained by applying this strategy.

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

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 [12].

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., $\mathit{idle}\ @\ t = \{t\}$), and *fail*, which always gives the empty set as its result ($\mathit{fail}\ @\ t = \emptyset$). 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: the concatenation combinator (;), in which $(S_1; S_2)\ @\ t = (S_1\ @\ t) \cup (S_2\ @\ t)$, the union combinator ($\vert$), where $(S_1 \vert S_2)\ @\ t = (S_1\ @\ t) \cup (S_2\ @\ t)$, and two forms of 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.

Several derived strategies may be defined using these combinators. For instance, the negation of a strategy S, written $\neg S$, which succeeds whenever S fails and vice versa, can be defined as $\neg S = S\ ?\ \mathit{fail} : \mathit{idle}$. The negation can be used to define the *normalization* strategy $!S$, which applies the strategy S

until it is no longer applicable: $S! = S^* \ ; \ \neg S$. [12]

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.

4 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 [27]. 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. The particular nature of a site determines what properties it possesses. For instance, a site could be a simple arithmetic function that takes two integers as parameters and returns their sum. In this case, the site has no side effects on its environment and a call to it should be placed in a context where an integer value is expected. Another site could be a deposit method of a bank account object that returns the new balance, in which case, firstly the site has a potential side effect on the state of the object, and secondly it might return different values at different invocations. A site could also be *time-sensitive*, hence returning different values at different times, such as a site responding with the current clock time value. A human user of an interactive system could also be modeled as a site from the point of view of an Orc program.

Sites per se are not part of the theory. Instead, they are assumed to exist, and the computations they provide constitute the data processed by Orc expressions. In general, nothing is assumed about sites beyond their existence and the fact that a site, when called, produces at most one value. A site may not respond to a call, either because it was originally designed to behave this way (for example a site that is designed to remain silent on a particular input), or due to a network or a server failure. 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 4, 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 ($> \ x \ >$); (2) the symmetric parallel composition operator ($|$), 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.

$let(x, y, \dots)$	A tuple constructor. Given a list of values, it returns a tuple consisting of these 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.

Table 2: Fundamental sites in Orc

In this section we present the syntax of the Orc language with some examples, and briefly discuss its asynchronous SOS semantics, which was first given in [10].

4.1 Syntax and Informal Semantics

A sequential programming language can be extended to include Orc computations by adding the following statement to its syntax:

$$z \ :\in \ E(L)$$

where E is an Orc expression name that must obviously have a defining equation (i.e., a declaration). Since we are interested in analyzing Orc programs in isolation, regardless of the host language, 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 [10]. The extended abstract syntax we use here is given in Table 3. In 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.

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 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

D	$\in$	$d_1; \ldots ; d_n$ (A list of declarations)
E	$\in$	Expression Name
x	$\in$	Variable
M	$\in$	Site
c	$\in$	Constant
h	$\in$	Handle
P	$\in$	$p_1, \ldots , p_n$ (A list of actuals)
Q	$\in$	$q_1; \ldots ; q_n$ (A list of formals)

Orc program	$::=$	D ; f
$d \in$ Declaration	$::=$	$E(Q) =_{def} f$
$f, g \in$ Expression	$::=$	$\mathbf{0} \mid M(P) \mid E(P)$
	$\mid$	$f \mid g$
	$\mid$	$f > x > g$
	$\mid$	f **where** $x :\in g$
	$\mid$	$!\, c \mid !\, x \mid ?\, h$
$p \in$ Actual Parameter	$::=$	$x \mid c \mid M$
$q \in$ Formal Parameter	$::=$	x

Table 3: Extended syntax of Orc

published by any one of them. Both sequential composition $f > x > g$ and asymmetric parallel composition g **where** $x :\in f$ *bind* the variable x in the expression g. An occurrence of a variable that is not bound by one of these composition operators is called a *free* occurrence. A sufficiently detailed informal description of the intended meaning of the three composition operators follows.

Symmetric parallel composition. An expression of the form $f \mid g$ asserts that f and g can be executed in parallel. A simple example is $\mathit{CNN} \mid \mathit{BBC}$, where *CNN* and *BBC* are sites that return the news as a value of some type. In this expression, the two sites are called concurrently. This may result in at most two responses being received. If either site (or both sites) fails to respond, the expression does not terminate. If both sites respond, the values published by the expression would be the values published by both sites in time order. In general, the sequence of values published by an expression $f \mid g$, when it terminates, is a time-ordered interleaving of the values published by f and g. For example, the expression $\mathit{let}(1) \mid \mathit{let}(2) \mid \mathit{let}(3)$ publishes any permutation of the the sequence $[1, 2, 3]$, under the assumption of termination.

Sequential composition. In an expression of the form $f > x > g$, f is executed first. If f does not publish a value, g is never executed, and the sequential composition does not terminate. A simple example is the expression $\mathit{if}(\mathit{false}) > x > \mathit{Email}(x, m)$, with $\mathit{Email}(x, m)$ a site that sends a fixed message m to user x. In this example, an e-mail message is never sent to any user, since $\mathit{if}(\mathit{false})$ is silent. On the other hand, if, while executing $f > x > g$, f publishes a value v, then g's substitution instance $g\{v/x\}$ is created and executed. For example, the expression $\mathit{lookup}(\mathit{user}) > x > \mathit{Email}(x, m)$, where $\mathit{lookup}(\mathit{user})$ is a site that returns the e-mail address of *user*, may result in sending the message m to the address returned. If $\mathit{lookup}(\mathit{user})$ responds with an address a, the value a is bound to x and $\mathit{Email}(a, m)$ is called. In this case, assuming $\mathit{Email}(a, m)$ responds with some value (like an error code), the value returned by $\mathit{Email}(a, m)$ is the value published by the sequential composition as a whole. If $\mathit{Email}(a, m)$ does not respond, evaluation of the sequential composition does not terminate and no value is ever published. Now suppose we have the expression $(\mathit{lookup}(\mathit{user}_1) | \mathit{lookup}(\mathit{user}_2)) > x > \mathit{Email}(x, m)$. The evaluation of this expression may result in up to two copies of m being sent, one to each of the addresses of user_1 and user_2. If both calls to *lookup* publish values and both calls to $\mathit{Email}(x, m)$ respond, the sequential composition publishes the two values returned by the calls to $\mathit{Email}(x, m)$ in time order.

The sequencing operator $> x >$ is also written as $\gg$ when no value passing takes place. In other words, the sequential composition $f > x > g$ can be written as $f \gg g$ if and only if x does not occur free in g.

Asymmetric parallel composition. The expression f **where** $x :\in g$ attempts to evaluate both expressions f and g concurrently. For instance, the expression $\mathit{Email}(x, m)$ **where** $x :\in \mathit{lookup}(\mathit{user})$ attempts to call both sites, but since site calls are strict, requiring concrete values for the call to be undertaken, only $\mathit{lookup}(\mathit{user})$ is called at this point. Once the $\mathit{lookup}(\mathit{user})$ call responds with a value v, $\mathit{Email}(v, m)$ is called, and if it in turn responds, execution terminates with this value being the result of the expression. Of course, if the expression was $\mathit{Email}(a, m)$ **where** $x :\in \mathit{lookup}(\mathit{user})$, then both sites would be called simultaneously, and if $\mathit{Email}(a, m)$ responds, then the value published by the expression is the value returned by $\mathit{Email}(a, m)$, regardless of whether $\mathit{lookup}(\mathit{user})$ responds or not. A more complex example would be one where g publishes more than one value. In this case, during the evaluation of g, once it publishes a value v, execution of g is terminated and the value v is assigned to x and is used in the evaluation of f. As an example, the expression

$$(\mathit{Email}(\mathit{admin}, m) \mid \mathit{Email}(x, m)) \textbf{ where } x :\in (\mathit{lookup}(\mathit{user}_1) \mid \mathit{lookup}(\mathit{user}_2))$$

causes the message m to be sent to an administrator and to at most one of the two users. If two e-mail messages are sent, the order in which they are sent is not determined and depends on which call to *Email* was first made. The above examples illustrate the fact that in an asymmetric parallel composition f **where** $x :\in g$, evaluation of the expression f may potentially block while waiting for a concrete value for x to be published by g.

4.2 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 [27]. We will refer to some of these expressions later in the report. 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.

$$M \mid \mathit{rtimer}(1) > x > M \mid \mathit{rtimer}(2) > x > M \mid \mathit{rtimer}(3) > x > M$$

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.

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

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:

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

Site M is given priority over site N, in that a response from M, if received within 1 time unit, would be the value published by the expression. Otherwise, either value published by M or N is published.

If we assume a site $or(b_1, b_2)$ that publishes the logical *or* of its two boolean arguments, then by the strictness of site calls, the booleans b_1 and b_2 have to be evaluated to truth values before the site call is made. The following program implements a non-strict *Or*, and uses it to evaluate a simple logical formula.

$$\begin{array}{l} \mathit{ift}(a) =_{def} \mathit{if}(a) > t > \mathit{let}(a); \\ \mathit{Or}(x; y) =_{def} \mathit{let}(z) \textbf{ where } z :\in \mathit{ift}(x) \mid \mathit{ift}(y) \mid \mathit{or}(x, y); \\ \mathit{Or}(x, y) \\ \quad \textbf{where } x :\in \mathit{eq}(2, 2) \\ \quad \textbf{where } y :\in \mathit{lt}(3, 1) \end{array}$$

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

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

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

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

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

$$!c \overset{!c}{\hookrightarrow} \mathbf{0}\ \ (\textsc{Pub}) \qquad \frac{E(Q) =_{def} f \in D}{E(P) \overset{\tau}{\hookrightarrow} f\{P/Q\}}\ (\textsc{Def}) \qquad \frac{f \overset{!c}{\hookrightarrow} f'}{g \textbf{ where } x :\in f \overset{\tau}{\hookrightarrow} g\{c/x\}}\ (\textsc{Asym1V})$$

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

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

Figure 1: Asynchronous semantics of Orc

4.3 Asynchronous Structural Operational Semantics of Orc

The asynchronous operational semantics introduced in [10] (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 2.3 of [10]. 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. In order to rule out such undesirable behaviors, a *synchronous semantics* is proposed in [10] 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 $\hookrightarrow$ 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. 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 two alternative ways: (i) by means of Maude's strategy languages; and (ii) by a rewriting semantics with additional equational conditions that requires no strategies.

5 SOS-Based 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 5.3 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 5.4). 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 [10] the evaluation of an expression may go through several quiescent states, followed by processing of new internal events after each external event reception.

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 report, except where otherwise indicated.

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

5.1 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.

$$\begin{aligned}
\_\langle\_, \_|\_\rangle &: \mathit{SiteName} \times \mathit{ConstList} \times \mathit{Handle} \times \mathit{Nat} \to \mathit{Event}\\
\_?\_|\_ &: \mathit{Handle} \times \mathit{Const} \times \mathit{Nat} \to \mathit{Event}\\
!\_|\_ &: \mathit{Const} \times \mathit{Nat} \to \mathit{Event}\\
\tau &: \to \mathit{Event}
\end{aligned}$$

The timed event $M\langle C, h|n\rangle$ 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 $h?c|n$. The third event operator, $!c|n$, denotes publishing a value c at time n, and, finally, τ is an 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*.

$$h : \mathit{Nat} \to \mathit{Handle}$$

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*).

$$\begin{array}{lcl} mt & : & \rightarrow \mathit{Context} \\ \_,\_ & : & \mathit{Context} \times \mathit{Context} \rightarrow \mathit{Context} \end{array}$$

Initially, a context is created out of the declaration list of an Orc program (see Table 5) 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 [21] [*ConstList*] and [*Msg*] are used instead of the sorts *ConstList* and *Msg*.

$$[\_,\_,\_] : \mathit{SiteName} \times [\mathit{ConstList}] \times \mathit{Handle} \rightarrow [\mathit{Msg}]$$

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*.

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

Configurations. An Orc configuration constitutes a state of the system.

A *configuration* consists of an Orc expression and a record.

$$\langle \_, \_ \rangle : \mathit{Expr} \times \mathit{Record} \to \mathit{Conf}$$

A *record* is a set of fields (built with an associative-commutative set union operator $\_|\_$), 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* $(clk : c_n)$, the *pool of messages* $(msg : \rho)$, and the *next available handle name* $(hdl : h)$. As explained in Section 3.2, besides configurations $\langle f, r \rangle$, we also allow variants $\{f, r\}$ and $[f, r]$ to model single-step rewrites.

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

5.2 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.

$$\begin{aligned}\textsc{SiteCall} : \{M(C), tr : t \mid msg : \rho \mid hdl : h_n \mid clk : c_m \mid r\} \\ \to [?h_n, tr : t \;.\; M\langle C, h_n | m\rangle \mid msg : \rho \; [M, C, h_n] \mid \\ hdl : h_{n+1} \mid clk : c_m \mid r]\end{aligned}$$

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 $(?h)$, where h 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.

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

$$[\mathit{self}, \mathit{app}(M, C, \mathit{rand}), h_n],$$

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. The operator *rand* simulates a pseudo-random integer between 1 and 10 inclusive using the following rules:

$$\textsc{Rand} : \mathit{rand} \to \mathit{floor}((\mathit{random}(\mathit{counter})/4294967296) \times 10)$$

$$\textsc{Count} : counter \rightarrow s(counter) \quad \textsc{Eval} : counter \rightarrow 0$$

with *random* an internal pseudo-random number generator[1] and s the successor function. The operational meaning is 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.

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

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.

$$\textsc{Pub} : \{!c, tr : t \mid clk : c_m \mid r\} \rightarrow [\mathbf{0}, tr : t\ .\ (!c|m)|clk : c_m \mid r]$$

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.

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

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-time[2]. 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

[1]The function *random* is defined in Maude in the module `RANDOM` (see [9]). Given a seed c, *random*(c) generates a random number between 0 and 4294967296

[2]Substituting one variable at-a-time does not pose a problem as it is here equivalent to being done simultaneously. A variable can only be substituted with a variable, a constant, or a site name. Moreover, whenever renaming is needed to avoid free variable capture, CINNI reflects this renaming in the substitution term, keeping track of the right substitution (see sections 8.1 and 8.2).

published by the evolution of f, a new instance $g\{c/x\}$ of $g(x)$ is created.

$$\textsc{Seq1V}: \{f > x > g, tr : t \mid r\} \rightarrow [(f' > x > g)|g\{c/x\}, tr : t \,.\, \tau \mid r'] \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : (!c|m))|r']$$

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, one rule for each type of event. These are the rules labeled Seq1N1, Seq1N2, and Seq1N3, and shown below.

$$\textsc{Seq1N1}: \{f > x > g, tr : t \mid r\} \rightarrow [f' > x > g, tr : t \,.\, \tau \mid r'] \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : \tau \mid r']$$

$$\textsc{Seq1N2}: \{f > x > g, tr : t \mid r\} \rightarrow [f' > x > g, tr : t \,.\, h?c|m \mid r'] \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : h?c|m \mid r']$$

$$\textsc{Seq1N3}: \{f > x > g, tr : t \mid r\} \rightarrow [f' > x > g, tr : t \,.\, M\langle C, h|m\rangle \mid r'] \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : M\langle C, h|m\rangle \mid r']$$

The rules simply allow f to evolve while recording the appropriate event in the execution trace.

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 $(|)$ is assumed associative and commutative, only one instance of the rule (labeled Sym) is required.

$$\textsc{Sym}: \{f \mid g, tr : t \mid r\} \rightarrow [f'|g, tr : t \,.\, L \mid r'] \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : L|r']$$

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.

$$\textsc{Asym1V}: \{g \textbf{ where } x :\in f, tr : t \mid r\} \rightarrow [g\{c/x\}, tr : t \,.\, \tau \mid r'] \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : (!c|m) \mid r']$$

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 f are specified by the rules labeled Asym1N1, Asym1N2 and Asym1N3, shown below.

$$\textsc{Asym1N1}: \{g \textbf{ where } x :\in f, tr : t \mid r\} \rightarrow [g \textbf{ where } x :\in f', tr : t \,.\, \tau \mid r'] \quad \text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : \tau \mid r']$$

$$\textsc{Asym1N2}: \{g \ \mathbf{where}\ x :\in f, tr : t \mid r\} \\ \to [g \ \mathbf{where}\ x :\in f', tr : t \ . \ h?c|m \mid r'] \\ \text{if } \{f, tr : nil \mid r\} \to [f', tr : h?c|m \mid r']$$

$$\textsc{Asym1N3}: \{g \ \mathbf{where}\ x :\in f, tr : t \mid r\} \\ \to [g \ \mathbf{where}\ x :\in f', tr : t \ . \ M\langle C, h|m\rangle \mid r'] \\ \text{if } \{f, tr : nil \mid r\} \to [f', tr : M\langle C, h|m\rangle \mid r']$$

Finally, g is also allowed to evolve while generating any kind of event. This is specified by the following rule:

$$\textsc{Asym2}: \{g \ \mathbf{where}\ x :\in f, tr : t \mid r\} \to [g' \ \mathbf{where}\ x :\in f, tr : t \ . \ L \mid r'] \\ \text{if } \{g, tr : nil \mid r\} \to [g', tr : L \mid r']$$

The key point about the above rewriting Orc semantics is that it *faithfully mirrors* the SOS Orc semantics from [10] 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 our theory described above, 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 3.2 and described in full detail in [23]; and (iii) by Theorem 1 in [23], 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 the rewrite theory given above the rules labeled Count, Eval, and Rand added for execution purposes.

5.3 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 ready to be made are waiting. In what follows, we describe how, in agreement with the synchronous semantics of [10], 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.

$$in \ := \ \textsc{SiteCall} \mid \textsc{Pub} \mid \textsc{Def} \mid \textsc{Sym}[in] \mid \textsc{Seq}[in] \mid \textsc{Asym}[in]$$

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.

$$ex := \textsc{SiteRet} \mid \textsc{Sym}[ex] \mid \textsc{Seq}[ex] \mid \textsc{Asym}[ex]$$

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

$$sync := \textsc{Step}[in\ ?\ idle : ex]^{+}$$

where Step is the label of the step rule described in Section 3.2, 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 [10]). Otherwise, the external strategy is attempted on the original Orc configuration (corresponding to a $\hookrightarrow_R$ step in the sense of [10]). If both substrategies fail, the condition is not satisfied and thus the step rule is not taken. We shall refer to the rewrite theory developed so far that implements the synchronous execution strategy by $\mathcal{R}^{sos}_{Orc}$. In the next section, we will show how $\mathcal{R}^{sos}_{Orc}$ can be extended to $\mathcal{R}^{sos}_{Orc}$ that supports the timed semantics of Orc.

5.4 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 [10], 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 in the theory $\mathcal{R}^{sos}_{Orc}$. For this purpose, as usual for rewriting logic semantic definitions of real-time systems [30], we use a (discrete) time domain (maintained by the *clk* field in a configuration), and a "tick" rewrite rule to advance time:

$$\textsc{Tick} : \{f, clk : c_n \mid r\} \to [f, clk : c_{n+1} \mid \delta(r)]$$

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.

5.4.1 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 4.2, 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.[3]

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.

[3]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.

The Strategy Language Approach: To achieve the behavior described above, we can easily extend the strategy expression *sync-instant* given in Section 5.3 for instantaneous transitions, so that the tick rule is taken into account . The new strategy is

$$\textit{sync-timed} := \textsc{Step}[\textit{in} \; ? \; \textit{idle} : (\textit{ex} \; ? \; \textit{idle} : \textsc{Tick})]^{+}$$

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 specified by the following expression:

$$\textit{timed} := \textsc{Step}[\textit{eager} \; ? \; \textit{idle} : \textsc{Tick}]^{+}$$

where *eager* is defined as

$$\textit{eager} \quad := \quad \textsc{SiteCall} \mid \textsc{Pub} \mid \textsc{Def} \mid \textsc{SiteRet} \mid \textsc{Sym} \mid \textsc{Seq} \mid \textsc{Asym}$$

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:

$$\textsc{Tick} : \{f, \textit{clk} : c_n \mid r\} \to [f, \textit{clk} : c_{n+1} \mid \delta(r)]$$
$$\text{if } \textit{eagerEnabled}(\{f, \textit{clk} : c_n \mid r\}) \neq \textit{true}$$

The predicate *eagerEnabled* is defined using a technique similar to the one proposed in [31]. 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 [31], 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,

$$\textit{eagerEnabled} \; : \; \textit{Conf} \to \; \texttt{[Bool] [frozen]}$$

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 $\{f, r\} \to$

$[f', r']$ if $C \wedge \bigwedge_{i=1}^{n} \{f_i, r_i\} \rightarrow [f'_i, r'_i]$ 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

$$\mathit{eagerEnabled}(\{f, r\}) = \mathit{true} \text{ if } C \wedge \bigwedge_{i=1}^{n} \mathit{eagerEnabled}(\{f_i, r_i\})$$

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 Appendix B 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. In particular, the predicate *intAction* is first declared as follows

$$\mathit{intAction} \;:\; \mathit{Conf} \rightarrow \texttt{[Bool] [frozen]}$$

Then, for each rule in $\mathcal{R}_{Orc}^{sos}$ other than the SiteRet rule, $\{f, r\} \rightarrow [f', r']$ if $C \wedge \bigwedge_{i=1}^{n} \{f_i, r_i\} \rightarrow [f'_i, r'_i]$, we introduce an equation

$$\mathit{intAction}(\{f, r\}) = \mathit{true} \text{ if } C \wedge \bigwedge_{i=1}^{n} \mathit{intAction}(\{f_i, r_i\})$$

Finally, since a site return may occur in a subexpression of a composed expression f, we subject the application of a rule applying a site return transition to the condition that f cannot take an internal action (i.e., the *intAction* predicate is not true). In other words, we have the rules

$$\begin{aligned} r : \{f \oplus g, tr : t \mid r\} \;&\rightarrow [f' \oplus g, tr : t \;.\; (h?c|n) \mid r'] \\ &\text{if } \{f, tr : nil \mid r\} \rightarrow [f', tr : (h?c|n)|r'] \\ &\qquad \wedge\, \mathit{intAction}(\{f \oplus g, tr : t \mid r\}) \neq \mathit{true} \end{aligned}$$

where $\oplus$ stands for an Orc composition operator, and similarly for g. We denote by $\mathcal{R}_{Orc}^{sos}$ the rewrite theory specifying the synchronous, real-time semantics of Orc using both the *eagerEnabled* and *intAction* predicates.

6 Reduction Rewriting Orc Semantics

A serious operational drawback of the SOS-based semantics, $\mathcal{R}^{sos}_{Orc}$, introduced in the previous section is the excessive number of rewrite rules having rewrites in their conditions, which is typical of the SOS specification style. A rewrite, as opposed to an equation, is usually expensive to find a proof for or to disprove as it is non-deterministic in nature. This non-determinism stems from the fact that rewrite rules may introduce new variables that could typically be matched in many different ways. In addition, the multiplicity of such rules in the specification can potentially cause nested (recursive) rewrite checks when checking a rule's conditions that adversely affect performance of execution and analysis.

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 the previous section, 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 more flexible semantics and a potentially smaller state space, since only transitions caused by rules appear in the state space. The proposed specification, which we will henceforth call $\mathcal{R}^{red}_{Orc}$, is still operational, in that it describes in detail how Orc programs are evaluated, and is, in fact, equivalent to $\mathcal{R}^{sos}_{Orc}$, in the sense that, given any Orc program P, the state transition systems of the semantics of P given by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ are strongly bisimilar, assuming that program configurations are closed[4]. However, by reducing the number of rewrites and their complexity we achieve a simpler and indeed superior semantic specification that can be executed and analyzed much more efficiently. In this formalization, only non-deterministic aspects of Orc are modeled by rewrite rules, which are mostly unconditional. Orc's deterministic features are specified in a more abstract and efficient way with equations.

6.1 Semantic Infrastructure

The data structures and operations needed for a smooth development of the semantic definitions of the language are similar to those of the SOS-based specifications, with a few exceptions. We list the three major differences below:

1. Unlike for the SOS-based semantics, events (or actions) are not essential to $\mathcal{R}^{red}_{Orc}$ and so one could remove them altogether. However, to maintain equivalence with $\mathcal{R}^{sos}_{Orc}$ and to provide a simple means by which we can compare the semantics of programs of both models, we choose to keep events and include traces as part of Orc configurations.

2. Unlike $\mathcal{R}^{sos}_{Orc}$, where rewrites occur among terms representing Orc configurations, rewrites in $\mathcal{R}^{red}_{Orc}$ may occur at the level of Orc expressions or

[4]Roughly speaking, a configuration $\langle f, r\rangle$ is *closed* if every expression name referenced in f has a declaration in r. This is discussed in more detail when we introduce the equivalence theorem in Section 7.

subexpressions. This entails an appropriate change to the frozenness properties of the syntactic representations of the sequential composition operator. In $\mathcal{R}^{red}_{Orc}$, we declare the sequential composition expression $f > x > g$ to be *frozen* on its third argument g, that is, we define $\phi(\_ > \_ > \_) = \{3\}$.

3. The strategy we used in the SOS-based semantics to implement one-step rewrites is clearly no longer needed. This simplifies the specification even further by eliminating the two configuration constructors introduced by the MSOS-to-rewriting logic transformation method and the one-step rewrite rule. However, rewrites at the expression level still need to be controlled so that the single-step operational semantics is correctly captured. For this purpose, the configuration operator $\langle f, r\rangle$ is declared *frozen* on its first argument, i.e., $\phi(\langle \_, \_\rangle) = \{1\}$.

6.2 Orc Rewriting Semantics Specification

One can specify how an Orc program evolves at different levels of abstraction. As was explained in the introduction, rewriting logic facilitates this flexibility by providing equational definitions to abstract away deterministic aspects of the behavior of a concurrent system, while truly concurrent features of interest are modeled with rewrite rules. To provide a fair basis for comparison with the semantic specifications of the previous section, we choose a level of abstraction that captures precisely the same features given by the SOS-based specifications.

We now specify a rewrite $\mathcal{R}^{red}_{Orc} = (\Sigma, E, R, \phi)$ theory that 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. For each of these actions, except the site return action which is modeled by a single rewrite rule, two rewrite rules are used: one rule is conditional at the Orc configuration level, and the other is unconditional at the expression level. The first rule defined at the configuration level is used to enforce the fact that during each rewrite step of the configuration, exactly one action takes place within the underlying expression. An error term of the *kind* [*Expr*] is produced in the case of unwanted interleaving of such actions.

$$err : \to [Expr]$$

To guarantee confluence of the equations E, an expression having *err* as a subterm immediately collapses to *err*, using the following equations,

$$\begin{aligned}
&err \mid W = err\\
&err > x > W = err\\
&W > x > err = err\\
&err \textbf{ where } x :\in W = err\\
&W \textbf{ where } x :\in err = err
\end{aligned}$$

with W a variable of the kind [*Expr*]. In the following paragraphs, we describe the rules R and the rest of the equations in E. The complete specifications in Maude can be found in Appendix A.1.6.

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

$$\begin{array}{rcl}\textsc{SiteCall} & : & \langle f, r\rangle \to \langle sc^{\uparrow}(f', M, C), r\rangle \;\text{ if }\; f \to sc^{\uparrow}(f', M, C)\\ \textsc{SiteCall*} & : & M(C) \to sc^{\uparrow}(\gamma, M, C)\end{array}$$

with γ a constant expression representing a temporary place holder expression. A site call subterm of an expression f rewrites to an operator $sc^{\uparrow}$

$$sc^{\uparrow} : \mathit{Expr}\ \mathit{SiteName}\ \mathit{ConstList} \to [\mathit{Expr}]$$

that propagates the call to the root of f, using the following equations,

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

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

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

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; (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, $sc^{\downarrow}$,

$$sc^{\downarrow} : \mathit{Expr}\ \mathit{Handle} \to \mathit{Expr}\ [\mathsf{frozen\ (1)}]$$

that traverses down the expression tree until it reaches the appropriate subterm where the handle is inserted.

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

Finally, the cases in which mutiple (internal) actions take place concurrently with a site call are taken care of with the following equations,

$$\begin{aligned}
sc^{\uparrow}(W, M, C) &= err \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
sc^{\uparrow}(f, M, C) \mid W &= err \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
sc^{\uparrow}(f, M, C) > x > W &= err \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
sc^{\uparrow}(f, M, C) \textbf{ where } x :\in W &= err \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
W \textbf{ where } x :\in sc^{\uparrow}(f, M, C) &= err \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
sc^{\downarrow}(W, h) &= err \quad \text{if} \quad W : \mathit{Expr} = \mathit{false}
\end{aligned}$$

The conditions state that the term substituted for W, which is of the kind $[\mathit{Expr}]$, cannot be reduced to one of the sort *Expr*, implying that an internal action has already taken place along with the site call action.

Expression Call. The specification of an expression call is similar to a site call, in that two operators,

$$\begin{aligned}
ec^{\uparrow} &: \mathit{Expr}\ \mathit{ExprName}\ \mathit{AParamList} \rightarrow [\mathit{Expr}] \\
ec^{\downarrow} &: \mathit{Expr}\ \mathit{Expr} \rightarrow \mathit{Expr}\ [\mathsf{frozen}]
\end{aligned}$$

are defined to propagate the call and its effect to and from the enclosing configuration. First, an expression call is modeled by the following two rewrite rules.

$$\begin{aligned}
\textsc{Def} &: \langle f, r\rangle \rightarrow \langle ec^{\uparrow}(f', E, P), r\rangle \ \text{ if } \ f \rightarrow ec^{\uparrow}(f', E, P) \\
\textsc{Def*} &: E(P) \rightarrow ec^{\uparrow}(\gamma, E, P)
\end{aligned}$$

Then, the call is propagated up the expression tree to the enclosing configuration, using the equations,

$$\begin{aligned}
ec^{\uparrow}(f_1, E, P) \mid f_2 &= ec^{\uparrow}(f_1 \mid f_2, E, P) \text{ if } f_2 \neq \mathbf{0}, \\
ec^{\uparrow}(f_1, E, P) > x > f_2 &= ec^{\uparrow}(f_1 > x > f_2, E, P), \\
ec^{\uparrow}(f_1, E, P) \textbf{ where } x :\in f_2 &= ec^{\uparrow}(f_1 \textbf{ where } x :\in f_2, E, P), \\
f_2 \textbf{ where } x :\in ec^{\uparrow}(f, E, P) &= ec^{\uparrow}(f_2 \textbf{ where } x :\in f_1, E, P).
\end{aligned}$$

where the effect of the call is recorded and the required declaration is accessed with the equation,

$$\begin{aligned}
&\langle ec^{\uparrow}(f, E, P), tr : t \mid con : (\sigma, E(Q) =_{def} g) \mid r\rangle \\
&\qquad\qquad = \langle ec^{\downarrow}(f, g\{P/Q\}), tr : t.\tau \mid con : (\sigma, E(Q) =_{def} g) \mid r\rangle
\end{aligned}$$

The resulting expression is propagated back to the appropriate subterm, using the $ec^{\downarrow}$ operator.

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

Finally, to account for unwanted concurrent executions, the following equations are used,

$$\begin{aligned}
ec^{\uparrow}(W, E, P) &= \mathit{err} \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
ec^{\uparrow}(f, E, P) \mid W &= \mathit{err} \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
ec^{\uparrow}(f, E, P) > x > W &= \mathit{err} \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
ec^{\uparrow}(f, E, P) \textbf{ where } x :\in W &= \mathit{err} \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
W \textbf{ where } x :\in ec^{\uparrow}(f, E, P) &= \mathit{err} \quad \text{if} \quad W : \mathit{Expr} = \mathit{false} \\
ec^{\downarrow}(W, g) &= \mathit{err} \quad \text{if} \quad W : \mathit{Expr} = \mathit{false}
\end{aligned}$$

Publishing a Value. Publishing a value is treated slightly differently from the previous internal actions. This is because the published value in a subterm of f may be bound to a variable and then consumed in f by a sequential or asymmetric parallel composition operator, in which case a τ event is propagated to the enclosing configuration. If the published value is not bound in f, a publish event is propagated up the tree all the way to the configuration.

The rewrite rules modeling the publishing of values are as follows.

$$\begin{aligned}
\textsc{Pub} &: \langle f, r\rangle \to \langle \mathit{pub}(f', c), r\rangle \ \text{ if } \ f \to \mathit{pub}(f', c) \\
\textsc{Pub}^{\tau} &: \langle f, r\rangle \to \langle \mathit{pub}^{\tau}(f'), r\rangle \ \text{ if } \ f \to \mathit{pub}^{\tau}(f') \\
\textsc{Pub}^{*} &: \ !c \to \mathit{pub}(\mathbf{0}, c)
\end{aligned}$$

In these rules, two operators are used:

$$\begin{aligned}
\mathit{pub} &: \mathit{Expr}\ \mathit{Const} \to [\mathit{Expr}] \\
\mathit{pub}^{\tau} &: \mathit{Expr} \to [\mathit{Expr}]
\end{aligned}$$

The operator *pub* propagates the published value up to a binding expression or to the enclosing configuration (whichever it reaches first), specified by the equations

$$\begin{aligned}
\mathit{pub}(f, c) \mid f' &= \mathit{pub}(f \mid f', c) \text{ if } f' \neq \mathbf{0} \\
\mathit{pub}(f, c) > x > g &= \mathit{pub}^{\tau}(f > x > g \mid g\{c/x\}) && (1) \\
\mathit{pub}(f, c) \textbf{ where } x :\in f' &= \mathit{pub}(f \textbf{ where } x :\in f', c) \\
g \textbf{ where } x :\in \mathit{pub}(f, c) &= \mathit{pub}^{\tau}(g\{c/x\}) && (2)
\end{aligned}$$

The equations 1 and 2 reflect, respectively, the semantics specified by the SOS rules Seq1V and Asym1V of Figure 1. They also transfer the propagation task to the second operator pub^{τ} in the cases where the published value is consumed by the expression. Otherwise, if the value is not consumed by the expression, *pub* eventually reaches the top and a publish event is recorded.

$$\langle \mathit{pub}(f, c), \mathit{tr} : t \mid \mathit{clk} : c_m \mid r\rangle = \langle f, \mathit{tr} : t.(!c|m) \mid \mathit{clk} : c_m \mid r\rangle$$

The operator pub^{τ}, which causes a τ event to be ultimately recorded if a published value is consumed by an expression, is specified by the equations

$$\begin{array}{rcl} pub^{\tau}(f) \mid f' & = & pub^{\tau}(f \mid f') \text{ if } f' \neq \mathbf{0} \\ pub^{\tau}(f) > x > f' & = & pub^{\tau}(f > x > f') \\ pub^{\tau}(f) \textbf{ where } x :\in f' & = & pub^{\tau}(f \textbf{ where } x :\in f') \\ f' \textbf{ where } x :\in pub^{\tau}(f) & = & pub^{\tau}(f' \textbf{ where } x :\in f) \\ \langle pub^{\tau}(f), tr : t \mid r \rangle & = & \langle f, tr : t.\tau \mid r \rangle \end{array}$$

Finally, error terms are captured by the following equations

$$\begin{array}{r} pub(W, c) = err \text{ if } W : Expr = false \\ pub(f, c) \mid W = err \text{ if } W : Expr = false \\ pub(f, c) > x > W = err \text{ if } W : Expr = false \\ pub(f, c) \textbf{ where } x :\in W = err \text{ if } W : Expr = false \\ W \textbf{ where } x :\in pub(f, c) = err \text{ if } W : Expr = false \\ pub^{\tau}(W) = err \text{ if } W : Expr = false \\ pub^{\tau}(f) \mid W = err \text{ if } W : Expr = false \\ pub^{\tau}(f) > x > W = err \text{ if } W : Expr = false \\ pub^{\tau}(f) \textbf{ where } x :\in W = err \text{ if } W : Expr = false \\ W \textbf{ where } x :\in pub^{\tau}(f) = err \text{ if } W : Expr = false \end{array}$$

Site Return. The following rule captures a site return.

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

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 $[self, c, h]$ is consumed, a site return event $h?c|m$ is generated, and the expression f is replaced with an operator sr

$$sr : Expr\ Const\ Handle \rightarrow Expr\ [\mathsf{frozen(1)}]$$

that propagates the return value down the expression tree to the appropriate pending call, using the following equations

$$\begin{array}{c} sr(f_1 \mid f_2, c, h) = sr(f_1, c, h) \mid sr(f_2, c, h) \text{ if } f_1 \neq \mathbf{0} \wedge f_2 \neq \mathbf{0}, \\ sr(f_1 > x > f_2, c, h) = sr(f_1, c, h) > x > f_2, \\ sr(f_1 \textbf{ where } x :\in f_2, c, h) = sr(f_1, c, h) \textbf{ where } x :\in sr(f_2, c, h), \\ sr(M(P), c, h) = M(P), \quad sr(\mathbf{0}, c, h) = \mathbf{0}, \quad sr(!x, c, h) = !x, \quad sr(!c', c, h) = !c', \\ sr(E(P), c, h) = E(P), \quad sr(?h', c, h) = \text{ if } (h = h') \text{ then } !c \text{ else } ?h', \end{array}$$

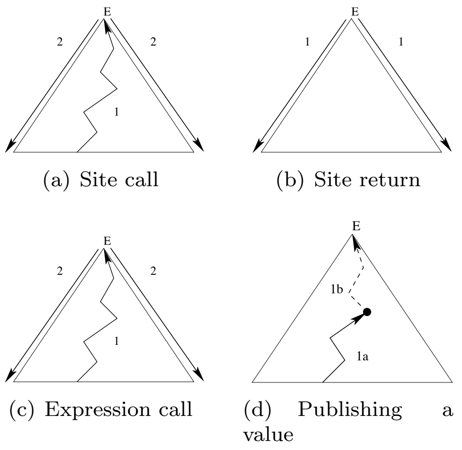

Figure 2: Schematic diagrams of the propagation of information in an expression just after the application of a rewrite rule

Error terms involving the *sr* operator are captured by the equation

$$sr(W, c, h) = err \text{ if } W : Expr = false$$

A common characteristic of the specifications of the four actions described above is the need to propagate information back and forth between a subterm of an expression and the configuration it is contained within. As we saw above, this propagation of information is specified using different axiliary functions that are defined inductively on the structure of an expression. Figure 2 gives a schematic representation of the mechanics of these basic actions. Figures 2(a) and 2(c) show that the structures of a site call and an expression call are similar, although the information propagated in both directions (and the side effects on the enclosing configurations) are different.

6.3 Instantaneous Execution Strategy

In accordance with the synchronous semantics of [27], we specify an execution strategy that gives the site return rule the least priority among the instantaneous actions. We denote by $\mathcal{R}^{red}_{Orc}$ the rewrite theory described above and extended to support the synchronous semantics of Orc. Here, again, we can use either Maude's strategy language or equational specifications to arrive at this $\mathcal{R}^{red}_{Orc}$. In this section we describe both approaches below.

Using Maude's strategy language, the simple strategy

$$sync^{red} = (\textsc{SiteCall} \mid \textsc{Pub} \mid \textsc{Pub}^{\tau} \mid \textsc{Def} \;?\; idle : \textsc{SiteRet})^{+}$$

achieves the desired synchronous behavior. We note that this strategy is much simpler than the corresponding strategy *sync* defined in Section 5.3 for $\mathcal{R}^{sos}_{Orc}$.

This is because rules in $\mathcal{R}^{red}_{Orc}$ have no rewrites in their conditions and the Step rule is avoided in $\mathcal{R}^{red}_{Orc}$.

Alternatively, the execution strategy can be specified equationally as follows. We first introduce the notion of an *active expression* as any expression having a site call, an expression call, or a publish expression as a sub-expression that could evolve. More precisely, we define the set of active expressions inductively as follows.

Definition 1. *The set of* active *expressions f_{active} in $\mathcal{R}^{red}_{Orc}$ is the smallest set given by the following rules.*

1. *$M(C)$, $E(P)$, and $!c$ are all in f_{active}.*
2. *$f \mid g \in f_{active}$ if $f \in f_{active}$ or $g \in f_{active}$.*
3. *$f > x > g \in f_{active}$ if $f \in f_{active}$.*
4. *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 that of a non-quiescent expression in [27]. This notion can be easily equationally captured by a predicate

$$active : Expr \rightarrow \texttt{[Bool] [frozen]}$$

defined by the equations

$$\begin{aligned}
active(f \mid f') &= active(f) \vee active(f') \text{ if } f \neq \mathbf{0} \wedge f' \neq \mathbf{0} \\
active(f > x > f') &= active(f) \\
active(f \textbf{ where } x :\in f') &= active(f) \vee active(f') \\
active(M(C)) &= true \\
active(!c) &= true \\
active(E(P)) &= true
\end{aligned}$$

The predicate is then used to limit the application of the SiteRet, as follows

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

6.4 Real-Time Semantics

So far we have only given an untimed semantics of Orc, that corresponds to the synchronous semantics of Section 2.3 in [27]. To specify the intended real-time semantics of Orc with the synchronous execution strategy of [27], we endow the specifications above with a discrete time clock and a "tick" rule, and implement a real-time synchronous strategy similar to the one given for the SOS-based specifications of $\mathcal{R}^{sos}_{Orc}$, where non-tick rules are executed eagerly, representing instantaneous transitions of an Orc configuration. The extended rewrite theory is denoted $\mathcal{R}^{red}_{Orc}$. To achieve this execution strategy, we can easily extend the strategy expression $sync^{red}$ to account for timing as follows

$$\textit{sync-timed}^{red} = (\textsc{SiteCall} \mid \textsc{Pub} \mid \textsc{Pub}^{\tau} \mid \textsc{Def} \mathbin{?} \mathit{idle} : (\textsc{SiteRet} \mathbin{?} \mathit{idle} : \textsc{Tick}))^{+}$$

which is, again, simpler than the corresponding strategy expression *sync-timed* for $\mathcal{R}^{sos}_{Orc}$ given in Section 5.4.1.

Alternatively, the execution strategy can be specified equationally in the following way. We first define an *eager* configuration as one that can make an instantaneous action.

Definition 2. *An Orc configuration $\mathcal{C}$ in $\mathcal{R}^{red}_{\mathrm{Orc}}$ is* eager *if $\mathcal{C}$ is of one of the following forms,*

1. $\langle f, r \rangle$ *with* $f \in f_{active}$.
2. $\langle f, \mathrm{msg} : \rho\, [\mathrm{self}, c, h] \mid r \rangle$ *if h is referenced in f.*

This notion of eager configurations can be easily captured by a predicate *eager*

$$\mathit{eager} \;:\; \mathit{Conf} \;\rightarrow\; \texttt{[Bool] [frozen]}$$

which evaluates to *true* if and only if it is applied to a configuration that can make an instantaneous action.

$$\begin{aligned} \mathit{eager}(\langle f, r \rangle) &= \mathit{true} \text{ if } \mathit{active}(f) \\ \mathit{eager}(\langle f, \mathit{msg} : \rho\, [\mathit{self}, c, h] \mid r \rangle) &= \mathit{true} \text{ if } h \in \mathit{handles}(f) \end{aligned}$$

Therefore, to capture the desired time-synchronous semantics in $\mathcal{R}^{red}_{Orc}$, we restrict the application of the tick rule by the condition that the configuration is not an eager configuration.

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

7 The Equivalence Theorem

We shall now show that the SOS-based rewriting semantics, $\mathcal{R}^{sos}_{Orc}$, and the reduction-based rewriting semantics, $\mathcal{R}^{red}_{Orc}$, are semantically equivalent, in the sense that an Orc program behaves in exactly the same way in both semantic models. We show this by proving a more general result, stating that the semantic models given by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ of any closed Orc configuration are strongly bisimilar. We first introduce what we mean by a configuration being closed (detailed proofs of most of the results in this section can be found in Appendix C).

Definition 3. *In a configuration* $\langle f, (\mathrm{con} : \sigma) \mid r\rangle$*, 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 4. *An Orc configuration* $\langle f, r\rangle$ *is* well-formed *if: (i)* f *does not contain any auxiliary function symbol, such as* $sc^{\uparrow}$*, pub, or* γ *(i.e.* f *is built using only the extended Orc syntax given in Table 3); and (ii)* r *contains at least the following five fields*

1. $\mathrm{tr} : t$*, with* t *an* EventList
2. $\mathrm{hdl} : h$*, with* h *a* Handle
3. $\mathrm{con} : \sigma$*, with* σ *a* Context
4. $\mathrm{msg} : \rho$*, with* ρ *a* MsgPool
5. $\mathrm{clk} : n$*, with* n *a* Nat.

Moreover, a closed *configuration is a well-formed configuration in which no expression name appears free in* f *or* σ*.*

We observe that a closed configuration in $\mathcal{R}^{sos}_{Orc}$ is also a closed configuration in $\mathcal{R}^{red}_{Orc}$ and vice versa. This is due to the fact that both $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ use the same semantic infrastructure. Moreover, we have the following easy lemma.

Lemma 1. *Let* $\mathcal{C}$ *be a closed configuration. If* $\mathcal{C} \rightarrow_{\mathcal{R}^{sos}_{\mathrm{Orc}}} \mathcal{C}'$ *for some configuration* $\mathcal{C}'$*, then* $\mathcal{C}'$ *is closed. Similarly, if* $\mathcal{C} \rightarrow_{\mathcal{R}^{red}_{\mathrm{Orc}}} \mathcal{C}'$*, then* $\mathcal{C}'$ *is closed.*

Proof. This can be proved by rule induction on the rewriting relations induced by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$, respectively. $\Box$.

Intuitively, preservation of well-formedness is trivial in both $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ by a quick examination of the rewrite rules. It is also easy to see that closed configurations in $\mathcal{R}^{sos}_{Orc}$ rewrite to configurations that are also closed. Essentially, the only rule in $\mathcal{R}^{sos}_{Orc}$ that might introduce expression names in an expression is the [Def] rule. But since all expression declarations are closed (have no free occurrences), and since actual parameters cannot be expression names, the resulting expression must also be closed. A similar argument applies to $\mathcal{R}^{red}_{Orc}$ to see that closed configurations are preserved in $\mathcal{R}^{red}_{Orc}$. In what follows, we assume all configurations are closed.

The following simple lemma proves a useful property of both rewrite theories. The property states that a list of events in the trace field of a configuration can always be prefixed with an arbitrary EventList without affecting the ways in which that configuration could later evolve. Intuitively, this is because rewrite rules and equations can match any EventList t, and when they do, t is either left unchanged or suffixed with a single event L.

Lemma 2. *For any* EventList s*, if* $\langle f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}^{sos}_{\mathrm{Orc}}} \langle f', \mathrm{tr} : t' \mid r'\rangle$ *then* $\langle f, \mathrm{tr} : s.t \mid r\rangle \rightarrow_{\mathcal{R}^{sos}_{\mathrm{Orc}}} \langle f', \mathrm{tr} : s.t' \mid r'\rangle$*. This property also holds in* $\mathcal{R}^{red}_{\mathrm{Orc}}$*.*

Next, we present some important properties of $\mathcal{R}_{Orc}^{red}$ that form an essential part of the proof of the main result of this section. They largely follow from the congruence inference rule of rewriting logic applied to expressions and configurations, and the assumption of well-formedness of configurations.

Lemma 3. *For any expression g,*

1. $\langle f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f', \mathrm{tr} : t.L \mid r'\rangle$ *if and only if*

$$\langle f \mid g, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f' \mid g, \mathrm{tr} : t.L \mid r'\rangle,$$

for L a site call event, a publishing event, or a τ event. If L is a site return event, then the equivalence holds provided that $active(g) \neq true$.

2. $\langle f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f', \mathrm{tr} : t.(!c|m) \mid r'\rangle$ *if and only if*

$$\langle f > x > g, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f' > x > g \mid g\{c/x\}, \mathrm{tr} : t.\tau \mid r'\rangle$$

3. *If L is not a publishing event, then*

$$\langle f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f', \mathrm{tr} : t.L \mid r'\rangle$$

if and only if

$$\langle f > x > g, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f' > x > g, \mathrm{tr} : t.L \mid r'\rangle$$

4. $\langle f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f', \mathrm{tr} : t.(!c|m) \mid r'\rangle$ *if and only if*

$$\langle g \text{ where } x :\in f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle g\{c/x\}, \mathrm{tr} : t.\tau \mid r'\rangle$$

5. *If L is a site call event or a τ event, then*

$$\langle f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle f', \mathrm{tr} : t.L \mid r'\rangle$$

if and only if

$$\langle g \text{ where } x :\in f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} g \text{ where } x :\in f', \mathrm{tr} : t.L \mid r'\rangle$$

If L is a site return event, then the equivalence holds provided that $active(g) \neq true$.

6. *If L is not a site return event,*

$$\langle g, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle g', \mathrm{tr} : t.L \mid r'\rangle$$

if and only if

$$\langle g \text{ where } x :\in f, \mathrm{tr} : t \mid r\rangle \rightarrow_{\mathcal{R}_{\mathrm{Orc}}^{red}} \langle g' \text{ where } x :\in f, \mathrm{tr} : t.L \mid r'\rangle$$

If L is a site return event, then the equivalence holds provided that $active(f) \neq true$.

The following lemma states that the definitions of eager configurations in $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ coincide.

Lemma 4. *For any configuration $\mathcal{C}$, $\mathcal{R}_{\text{Orc}}^{sos} \vdash \text{eagerEnabled}(\mathcal{C}) = \text{true}$ if and only if $\mathcal{C}$ is an eager configuration in $\mathcal{R}_{\text{Orc}}^{red}$.*

Similarly, the definitions of active configurations in $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ coincide. This is stated by the following lemma.

Lemma 5. *For any configuration $\mathcal{C}$, $\mathcal{R}_{Orc}^{sos} \vdash \text{intAction}(\mathcal{C}) = \text{true}$ if and only if $\mathcal{C}$ is an active configuration in $\mathcal{R}_{Orc}^{red}$.*

Now we are ready to present the main result of this section.

Theorem 1. *For any configurations $\mathcal{C}$ and $\mathcal{C}'$, the following equivalence holds,*

$$\mathcal{C} \rightarrow_{\mathcal{R}_{\text{Orc}}^{sos}} \mathcal{C}' \Longleftrightarrow \mathcal{C} \rightarrow_{\mathcal{R}_{\text{Orc}}^{red}} \mathcal{C}'.$$

The main result of this section can be derived as a consequence of Theorem 1 by taking as $\mathcal{C}$ the initial configuration of a program P given by $[P]$.

Corollary 1. *For any Orc program P and configuration $\mathcal{C}$, we have*

$$[P] \rightarrow_{\mathcal{R}_{Orc}^{sos}} \mathcal{C} \Longleftrightarrow [P] \rightarrow_{\mathcal{R}_{Orc}^{red}} \mathcal{C}.$$

Proof. Immediate from Theorem 1 and the fact that $[P]$ is closed, by definition. □

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

8 Specifications in Maude

In this section, we describe the specifications of both rewrite theories $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ in Maude. The full Maude specifications can be found in Appendix A and at the web site

`http://www.cs.uiuc.edu/homes/alturki/orc-maude`.

8.1 Specifying Names and Substitution

In the syntax given in Table 3 in Section 4.1 above, we distinguish four classes of names. These are: *site names*, *expression names*, *variable names*, and *handle names*. In Maude, a name can be represented as a quoted identifier, or a qid for short, which is a ground term of sort `Qid`. One way to represent classes of names is to define for each class a wrapper constructor that encapsulates a qid. Another approach, which we find more attractive, is to declare all name classes as subsorts of `Qid`. Then, using conditional membership equations, we specify the conditions under which a qid is of a particular class of names. This classification is based on the first letter (after the quote) of a qid, as follows. A qid starting with an ‘s’ is a site name, with an ‘e’ an expression name, and with a ‘v’ a variable name. Any other qid is considered an invalid name. Although this approach of representing classes of names might introduce additional checks while rewriting, it makes writing and reading programs a lot easier.

```
subsorts SiteName ExprName Var < Qid .
var Q : Qid .
cmb Q : SiteName if substr(string(Q), 0, 1) == "s" .
cmb Q : ExprName if substr(string(Q), 0, 1) == "e" .
cmb Q : Var if substr(string(Q), 0, 1) == "v" .
```

The keyword `cmb` above introduces a conditional membership axiom. The conditions use basic string manipulation functions defined in the Maude prelude. As described in Section 5.1, handle names, on the other hand, are represented differently using a wrapper over the natural numbers.

```
op h : Nat -> Handle .
```

This approach of defining names makes it easier to generate new handle names as required by the SiteCall rule.

Of the four classes of names mentioned above, only variable names may occur free or bound in an expression. Although expression names refer to expressions defined in the declarations, and thus they are in a sense bound by their defining equations, they are not passed as arguments and they cannot be communicated, unlike variable names[5]. In the syntax above, the only variable name binding constructs are the sequential and the asymmetric parallel composition operators. In an expression $f > x > g$, x is bound in g, whereas x is bound in f in the composition f **where** $x :\in g$. Otherwise, a variable name occurring in an expression f that is not bound by either of these operators is free.

To account for substitution of variable names with other variable names or constants, we use the CINNI calculus of explicit substitution [34]. This is consistent with our choice of first-order representation of Orc constructs in Maude. Thus, in specifying this particular instance of CINNI, we introduce indexed variable names, which represent free or bound occurrences of a variable (rather than binding occurrences of that variable which are represented by regular variable names).

```
op _{_} : Var Nat -> IVar [ctor prec 1] .
```

As done in [34], for a process calculus, we define three kinds of substitutions: the simple substitution `[_:=_]`, which accounts for substituting a name for a free name (assuming no free name capture), the shift-up substitution `[shiftup_]`, having the effect of substituting fresh names for free names, and finally the lift

[5]The notion of free and bound expression names is deferred to later sections, where it is more relevant. Here, for the purpose of defining substitution, we focus on free and bound variable names.

	`[`x `:=` a`]`	`[shiftup` a`]`	`[lift` x σ`]`
$x\{0\}$	a	$x\{1\}$	$x\{0\}$
$x\{1\}$	$x\{0\}$	$x\{2\}$	`[shiftup` x`]`$(\sigma\ (x\{0\}))$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x\{n\}$	$x\{n-1\}$	$x\{n+1\}$	`[shiftup` x`]`$(\sigma\ (x\{n-1\}))$
$y\{n\}$	$y\{n\}$	$y\{n\}$	`[shiftup` x`]`$(\sigma\ (y\{n\}))$

Table 4: CINNI's explicit substitution operators. The top row shows substitution expressions, which are being applied to the indexed variables in the leftmost column.

substitution `[lift__]`, which represents a more general substitution that avoids capturing free names. See the referenced paper for a detailed discussion of a similar instance of CINNI for the π-calculus.

```
op [_:=_] : Var AParam  -> Subst .
op [shiftup_] : Var -> Subst .
op [lift__] : Var Subst -> Subst .
```

The sort `Subst` is the sort of substitutions. The effect of these substitution operators on indexed variables is illustrated in Table 4, where we assume that $x \neq y$ and σ is a meta-variable ranging over terms of sort `Subst`. For instance, the following set of equations specify the lift substitution.

```
 eq [lift a S] a{0} = a{0} .
 eq [lift a S] a{s(n)} = [shiftup a] S a{n} .
ceq [lift a S] b{n} = [shiftup a] S b{n} if a =/= b .
```

The substitutions, as described above, are the same in every instance of CINNI because they do not depend on the particular language being specified. Once they are extended from variables to expressions in the language, the CINNI instance is distinguished from instances for other languages. In general, if the we denote by $\Uparrow_x S$ the lift substitution `[lift x S]`, a CINNI substitution is extended to language expressions by adding, for each syntactic constructor f of arity n in the language, an equation of the form,

$$S\ f(P_1,\ldots,P_n) = f(\Uparrow_{P_{j_1,1}} \cdots \Uparrow_{P_{j_1,m_1}} S\ P_1,\ldots,\Uparrow_{P_{j_n,1}} \cdots \Uparrow_{P_{j_n,m_n}} S\ P_n) \quad (3)$$

where each P_i is an expression in the language, and $P_{j_i,1},\ldots,P_{j_i,m_i}$ are the variable arguments that f binds in the ith expression argument P_i. This equation instantiated to Orc expressions along with examples are further discussed in the next section where the specification of Orc expressions in Maude are introduced.

8.2 Syntax Specifications and Examples

Building on the observations made above, we present the abstract syntax of Orc specified in Maude in Table 5.

```
fmod ORC-SYNTAX is
 op _;_ : DeclList Expr -> Prog [ctor prec 50] .

 op nilD : -> DeclList [ctor] .
 op _;_ : DeclList DeclList -> DeclList [ctor assoc id: nilD prec 40] .
 op __:=_ : ExprName FParamList Expr -> Decl [ctor frozen(3) prec 30] .

 op zero : -> Expr [ctor] .
 op _(_) : SiteName AParamList -> Expr [ctor prec 10] .
 op _(_) : ExprName AParamList -> Expr [ctor prec 10] .
 op !_ : IVar -> Expr [ctor prec 5] .
 op !_ : Const -> Expr [ctor prec 5] .
 op _>_>_ : Expr Var Expr -> Expr [ctor frozen(3) prec 15 gather (e & E)] .
 op _|_ : Expr Expr -> Expr [ctor assoc comm prec 20] .
 op _where_:in_ : Expr Var Expr -> Expr [ctor prec 25 gather (E & e)] .

 op ?_ : Handle -> Expr [ctor prec 1] .
endfm
```

Table 5: An excerpt from the functional module ORC-SYNTAX which specifies the extended syntax of Orc in Maude.

The special expression ? H, where H is a handle name, is not part of the original syntax but is required in the semantics rules of the language to represent a site call that is yet to return. The name H acts as a identifier for the call. Based on the algebraic properties of the Orc language constructs [10, 27], 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, as an equation (using the eq keyword), as follows.

```
eq zero > x > f = zero .
```

In addition to the operators of Table 5, a few syntactic sugar operators are defined below, where S and F are meta-variables of sorts SiteName and ExprName. respectively.

```
op _:=_ : ExprName Expr -> Decl [prec 30] .
eq E := f = E nilF := f .
op _() : SiteName -> Expr [prec 10] .
op _() : ExprName -> Expr [prec 10] .
eq M() = M(nilA) .
eq E() = E(nilA) .
```

Finally, the prec attribute, which appears in Table 5 and in the syntactic sugar operators above, specifies operator precedence; the lower the number associated with an operator, the higher the operator precedence. For instance, the expression

```
'sFormat('vn{0}) > 'vx > 'sDisplay('vx{0}) where 'vn :in 'sCNN() | 'sBBC()
```

is correctly parsed as

```
((('sFormat('vn{0})) > 'vx > ('sDisplay('vx{0})))
                                    where 'vn :in (('sBBC()) | ('sCNN()))
```

Using these syntactic specifications, the program PRIORITY given in Section 4.1, for instance, is represented as a term of the the theory `ORC-SYNTAX` as follows:

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

The CINNI calculus for explicit substitution is also extended to expressions. When a substitution is applied to an expression, the substitution propagates down the expression tree while keeping track of bound variable instances, so that the substitution can be correctly performed with no free variable capture. The substitutions are defined with equations that are instances of the equation 3 above. In our case, the most interesting CINNI equations are the sequential and the asymmetric parallel composition equations:

```
eq S (f > x > f') = (S f) > x > ([lift x S] f') .
eq S (f where x :in f') = ([lift x S] f) where x :in (S f') .
```

To illustrate how substitutions applied to expressions are performed, consider the expression

```
'eGetNews := 'sFormat('vn{0}, 'vp{0}) > 'vx > 'sDisplay('vx{0})
                 where 'vn :in 'sCNN()
```

We first note that `'vn{0}` and `'vx{0}` are bound in `'sFormat( 'vn{0}, 'vp{0} ) > 'vx > 'sDisplay( 'vx{0} )` and `'sDisplay('vx{0})`, respectively, while `'vp{0}` occurs free in `'eGetNews`. The zero index appearing in these occurrences means that if the variable is bound, then it is bound by the nearest binding occurrence of that variable. Now, suppose that we want to substitute `'va{0}` for `'vp{0}` in `'eGetNews`, that is, we want to evaluate `['vp := 'va{0}] 'eGetNews`. This substitution is simple (no free variable capture), and when applied to `'eGetNews` yields the expression

```
'sFormat('vn{0}, 'va{0}) > 'vx > 'sDisplay('vx{0}) where 'vn :in 'sCNN()
```

However, the substitution, `['vp := 'vn{0}]` has the potential of causing the free occurrence of `'vp{0}` to be captured by the binding occurrence of `'vn`. CINNI resolves this issue by renaming (shifting up the name of) the variable, giving the following expression as the result of the substitution.

```
'sFormat('vn{0}, 'vn{1}) > 'vx > 'sDisplay('vx{0}) where 'vn :in 'sCNN()
```

Note that `'vn{1}` is still free in the resulting expression. Finally, note that the substitutions `['vx := 'va{0}]` and `['vn := 'va{0}]` have no effect on `'eGetNews`, since there are no free occurrences of these variables in the expression. Other substitutions in which a variable is replaced by a constant value or a site name are simpler since neither values nor site names occur free or bound in an expression.

8.3 The Semantic Infrastructure and Rules

Since Maude supports a rich mixfix notation, the specifications of the semantic entities maintained in an Orc configuration and the rewrite rules defining the semantics of Orc are almost identical to those given in Section 5.1 and Section 5.2. Obviously, the Maude specification spells out all the necessary details for it to be complete. The reader is referred to Appendix A.1.3 for the Maude specifications of the semantic infrastructure, and to appendices A.1.5 and A.1.7 for the specifications of the synchronous instantaneous semantics given respectively by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$.

8.4 Real-Time Semantics and Site Definitions

As is explained in sections 5.4 and 6.4, the tick rule models the elapse of time in both rewrite theories $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$. In $\mathcal{R}^{sos}_{Orc}$, for instance, the tick rule is given as,

```
crl [Tick] :
  < f, (clk : clock(m)) | r > => < f, (clk : clock(s m)) | delta(r) >
                if eagerEnabled(< f, (clk : clock(m)) | r >) =/= true .
```

The *tick* rule advances time by one unit. The function `delta(r)` propagates the effect of a clock tick down the record structure. One such effect is updating time delays for messages[6].

```
eq delta([self,app(M, C, s(n)), h]) = [self, app(M, C, n), h] .
```

`delta(r)` also updates contents of messages containing relative timing information, such as messages from the $rtimer(t)$ site.

```
eq delta([self,app(rtimer, s(n), m), h]) = [self, app(rtimer, n, m), h] .
```

In order to be able to experiment with the above semantics of the Orc language and execute and verify 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 was achieved in part by defining the abstract application function `app()`. Then,

[6]Responses 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.

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 specification of a simple site module is that of the `if(b)` site shown below.

```
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 `PreConst` term that does not reduce to any ground `Const` term, mimicking a site not responding.

Sometimes, the definition of a site cannot be specified in isolation from its environment. For instance, the *clock* site needs to access the current clock time from the configuration. This can be easily done as follows.

```
eq < f , r | (msg : (rho [self, app(clock, nilA, n), h])) |
    (clk : clock(m)) > =
              < f , r | (msg : (rho [self, m, h])) | (clk : clock(m)) > .
```

To add some basic computational power to our Orc specification, we also define sites performing basic arithmetic functions, binary relations, and binary logical operations (See Appendix A.1.8).

9 Formal Analysis and Verification

In this section we illustrate how the real-time rewriting semantics we have developed can be used to experiment with Orc programs, explore traces of computations, and verify properties about them. By Theorem 1, we are at liberty to choose either $\mathcal{R}^{sos}_{Orc}$ or $\mathcal{R}^{red}_{Orc}$ to illustrate the rewriting semantics of Orc. As we shall see in Section 10, although the two theories are semantically equivalent, $\mathcal{R}^{red}_{Orc}$ is much more efficiently executable and analyzable. In this section, we focus on the use of Maude's features and tools in formally analyzing Orc programs and verifying properties about them. A performance comparison of the two specifications given by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ is deferred to the next section.

However, before we get to the examples, a vital observation is in order. Recall that time is kept track of using the `clock(m)` 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 generated random delays is the space of natural numbers and, thus, causes the state space of an Orc configuration to grow indefinitely. Of course, an Orc configuration having a recursive expression could be inherently an infinite-state system. However, for finite-state expressions, `clock(m)` and `counter` may cause even these finite-state systems to become infinite-state, which may severely limit our ability to analyze programs in the language. To counter these problems, we limit the time domain to a fixed number of clock ticks and limit the counter to a fixed subset of the natural numbers. These fixed numbers can be easily changed to better suit the example at hand by appropriately adjusting the following two lines,

```
eq clock(9) = halt .
```

and

```
eq s_^5(counter) = counter .
```

in the `CLOCK` and the `SIMCOUNTER` modules, respectively (see Appendix A.1.3). Using the specifications above, time might advance by ten clock ticks, and at most five pseudo-random numbers (between 0 and 10, inclusive) might be generated. For the following examples we shall assume the settings shown above, except where otherwise noted. Also, some portions of Maude's output are removed for presentation compactness.

9.1 Simple Examples

Site Calls and Time Delays. Assume *CNN* is a site that, when called, returns a signal (representing the news, as we are not really interested in the actual value returned). We want to examine a trace of a call to that site that successfully finishes. We can achieve that by issuing the following breadth-first search command in Maude (the number `1` in square brackets asks for exactly one solution, and the arrow `=>+` stands for one or more rewrites starting from the given configuration).

```
Maude> search [1] [nilD ; 'sCNN()] =>+ < zero , R:Record > .
Solution 1 (state 29)
states: 30  rewrites: 407 in 0ms cpu (12ms real) (~ rewrites/second)
R:Record --> (tr : ('sCNN < nilA,h(0) | 0 >) . (h(0) ? sig | 5) . !! sig |
  5) | (con : mt) | (clk : clock(5)) | (msg : empty) | hdl : h(1)
```

The command searches for a state of the system constructed by `[nilD ; 'sCNN()]` where there is nothing more to evaluate. The trace of this particular solution can be extracted from the `tr` indexed field, which is `('sCNN < nilA,'h | 0 >) . ('h ? sig | 5) . !! sig | 5)`. Note that the call event was made at time 0, while the response was not received or published until time 5. This implies that the response was delayed by 5 time units.

Consider now the *signal* site, which returns a signal immediately upon being called. Since responses of such site are insensitive to delays, they are received and published at the time the call is made. Thus, the trace of a simple call `signal()` is `(signal < nilA,'h | 0 >) . ('h ? sig | 0) . !! sig | 0` and appears in 10 states corresponding to the 10 clock ticks.

```
Maude> search [nilD ; signal()] =>+ < zero , R:Record > .
Solution 1 (state 8)
states: 9  rewrites: 64 in 0ms cpu (12ms real) (~ rewrites/second)
R:Record --> (tr : (signal < nilA,h(0) | 0 >) . (h(0) ? sig | 0) . !! sig
  | 0) | (con : mt) | (clk : clock(0)) | (msg : empty) | hdl : h(1)
...
```

up to

```
...
Solution 10 (state 18)
states: 19  rewrites: 242 in 10ms cpu (151ms real) (24200 rewrites/second)
R:Record --> (tr : (signal < nilA,h(0) | 0 >) . (h(0) ? sig | 0) . !! sig
  | 0) | (con : mt) | (clk : halt) | (msg : empty) | hdl : h(1)

No more solutions.
states: 19  rewrites: 242 in 10ms cpu (155ms real) (24200 rewrites/second)
```

Similarly, a call to the site `rtimer(t)` would result in the trace `(rtimer < t,'h | 0 >) . ('h ? sig | t) . !! sig | t`, where `t` is a natural number less than 10 (the maximum clock time), which appears in 10 - t states.

A site call might not return a response. A simple example is the call `if(tr(false))`. Since the state space of this call is finite, we can verify this property using the breadth-first search command of Maude.

```
Maude> search [nilD ; if(tr(false))] =>+ < zero , R:Record > .
No solution.
states: 38  rewrites: 626 in 10ms cpu (17ms real) (62600 rewrites/second)
```

Compositions. Next we consider a simple sequential composition expression. We look for a state where execution terminates and the value is published.

```
Maude> search [1] [nilD ; let(2) > 'vx > rtimer('vx{0})] =>+
  < zero , R:Record > .
Solution 1 (state 25)
states: 26  rewrites: 283 in 0ms cpu (10ms real) (~ rewrites/second)
R:Record --> (tr : (let < 2,h(0) | 0 >) . (h(0) ? 2 | 0) . tau . (rtimer
  < 2,h(1) | 0 >) . (h(1) ? sig | 2) . !! sig | 2) | (con : mt) | (clk :
  clock(2)) | (msg : empty) | hdl : h(2)
```

the first `tau` event was generated as a result of applying the SEQ1V rule, which creates a new instance of `rtimer('vx{0})` with `'vx{0}` bound to 2. Of course, if we try to find a similar state but with a clock time less than 2, the search fails.

```
Maude> search [1] [nilD ; let(2) > 'vx > rtimer('vx{0})] =>+
  < zero , (clk : clock(1)) | R:Record > .
No solution.
states: 34  rewrites: 464 in 0ms cpu (25ms real) (~ rewrites/second)
```

Recall that in a sequential composition $f > x > g$, there are as many new instances of g created as values published by f. To illustrate this point, consider the following expression, where we look for states during the first two clock ticks and in which execution is completed [7]. There are four solutions for each clock tick value of 2 and 3. The solutions differ only in the order of the actions taken. Only two solutions are shown below.

```
Maude> search [nilD ; (! 2 | ! 1) > 'vx > rtimer('vx{0})] =>+
  < zero , (clk : clock(N:Nat)) | R:Record > such that N:Nat < 4 .

Solution 1 (state 251)
states: 252  rewrites: 4011 in 60ms cpu (79ms real)
  (66850 rewrites/second)
R:Record --> (tr : tau . (rtimer < 1,h(0) | 0 >) . tau . (rtimer < 2,h(1)
  | 0 >) . (h(0) ? sig | 1) . (!! sig | 1) . (h(1) ? sig | 2) . !! sig |
  2) | (con : mt) | (msg : empty) | hdl : h(2)
N:Nat --> 2
...
Solution 5 (state 280)
states: 281  rewrites: 4765 in 90ms cpu (294ms real)
  (52944 rewrites/second)
R:Record --> (tr : tau . (rtimer < 1,h(0) | 0 >) . tau . (rtimer < 2,h(1)
  | 0 >) . (h(0) ? sig | 1) . (!! sig | 1) . (h(1) ? sig | 2) . !! sig |
  2) | (con : mt) | (msg : empty) | hdl : h(2)
N:Nat --> 3
...
No more solutions.
states: 324  rewrites: 6284 in 120ms cpu (377ms real)
  (52366 rewrites/second)
```

Symmetric parallel execution was also in a sense illustrated by the previous example as the new instances created execute in parallel with the evolved expression. The system state could get very large quickly as the number of expressions running in parallel increases. For example, the following modest expression has 1,036 states, 6 states of which represent terminating configurations in which evaluation terminates with the *zero* expression (the symbol `=>!` specifies that the resulting state must be a terminal state).

```
Maude> search [nilD ; atimer(1) | rtimer(2) | let(5)] =>!
  < zero , R:Record > .

Solution 1 (state 1030)
states: 1036  rewrites: 27674 in 240ms cpu (306ms real)
  (115308 rewrites/second)
R:Record --> (tr : (let < 5,h(0) | 0 >) . (atimer < 1,h(1) | 0 >) .
  (rtimer < 2,h(2) | 0 >) . (h(0) ? 5 | 0) . (!! 5 | 0) . (h(1) ? sig |
  1) . (!! sig | 1) . (h(2) ? sig | 2) . !! sig | 2) | (con : mt) |
```

[7] In this example, and in some examples to come, we use the internal publish expression $!\ c$ to skip over details of the site call $let(c)$.

```
  (clk : halt) | (msg : empty) | hdl : h(3)
...
Solution 6 (state 1035)
states: 1036  rewrites: 27674 in 280ms cpu (589ms real)
  (98835 rewrites/second)
R:Record --> (tr : (rtimer < 2,h(0) | 0 >) . (atimer < 1,h(1) | 0 >) .
  (let < 5,h(2) | 0 >) . (h(2) ? 5 | 0) . (!! 5 | 0) . (h(1) ? sig | 1)
  . (!! sig | 1) . (h(0) ? sig | 2) . !! sig | 2) | (con : mt) | (clk :
  halt) | (msg : empty) | hdl : h(3)

No more solutions.
states: 1036  rewrites: 27674 in 280ms cpu (595ms real)
  (98835 rewrites/second)
```

Next, consider the following expression where *atimer* is called with either the value 1 or 2, depending on which expression publishes its value first, using asymmetric parallel composition.

```
Maude> search [nilD ; atimer('vx{0}) where 'vx :in (! 1 | ! 2)] =>+
  < zero , (clk : clock(N:Nat)) | R:Record > such that N:Nat < 3 .

Solution 1 (state 23)
states: 24  rewrites: 248 in 10ms cpu (10ms real)
  (24800 rewrites/second)
R:Record --> (tr : tau . (atimer < 1,h(0) | 0 >) . (h(0) ? sig | 1) .
  !! sig | 1) | (con : mt) | (msg : empty) | hdl : h(1)
N:Nat --> 1

Solution 2 (state 29)
states: 30  rewrites: 346 in 20ms cpu (81ms real)
  (17300 rewrites/second)
R:Record --> (tr : tau . (atimer < 1,h(0) | 0 >) . (h(0) ? sig | 1) .
  !! sig | 1) | (con : mt) | (msg : empty) | hdl : h(1)
N:Nat --> 2

Solution 3 (state 32)
states: 33  rewrites: 408 in 20ms cpu (120ms real)
  (20400 rewrites/second)
R:Record --> (tr : tau . (atimer < 2,h(0) | 0 >) . (h(0) ? sig | 2) .
  !! sig | 2) | (con : mt) | (msg : empty) | hdl : h(1)
N:Nat --> 2

No more solutions.
states: 48  rewrites: 753 in 30ms cpu (128ms real)
  (25100 rewrites/second)
```

There are three such states during the first three clock ticks. Either the value 1 is published first and used to evaluate *atimer*, which returns a signal at time 1 and remains at that state for another clock tick, or the value 2 is published and a signal is returned at time 2.

9.2 Simulating Orc Programs and Verifying Simple Properties

Using Maude's `rewrite` command, Orc programs can be simulated to obtain possible runs. We can also use Maude's `search` command to check violations of invariants. These two aspects of formal analysis in Maude are illustrated below.

Consider the program TIMEOUT given in Section 4.2. 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: 528 in 0ms cpu (5ms real) (~ 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 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: 542 in 0ms cpu (5ms real) (~ 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.

Recall the TIMED-MCALL program presented in Section 4.1, in which a site M is called three times, in intervals of one time unit, starting immediately. We first simulate a run of the program using the rewrite command of Maude, which results in a configuration whose trace satisfies the specifications of the program.

```
Maude> rew [nilD ; 'sM() | rtimer(1) > 'vx > 'sM() |
  rtimer(2) > 'vx > 'sM() | rtimer(3) > 'vx > 'sM()] .
rewrites: 1482 in 10ms cpu (9ms real) (148200 rewrites/second)
result Conf: < zero,(tr : ('sM < nilA,h(0) | 0 >) . (rtimer < 1,h(1) | 0 >)
  . (rtimer < 2,h(2) | 0 >) . (rtimer < 3,h(3) | 0 >) . (h(1) ? sig | 1) .
  tau . ('sM < nilA,h(4) | 1 >) . (h(2) ? sig | 2) . tau . ('sM < nilA,h(5)
  | 2 >) . (h(3) ? sig | 3) . tau . ('sM < nilA,h(6) | 3 >) . (h(0) ? 1 |
  5) . (!! 1 | 5) . (h(4) ? 1 | 6) . (!! 1 | 6) . (h(5) ? 1 | 7) . (!! 1 |
  7) . (h(6) ? 1 | 8) . !! 1 | 8) | (con : mt) | (clk : halt) | (msg :
  empty) | hdl : h(7) >
```

We now verify that the property that no two calls to M occur at the same time is satisfied in any state to which the program could evolve. This can be achieved by issuing the following search command, for which no solution exists, as required (the notation `=>*` 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: 181865  rewrites: 7961763 in 71370ms cpu (75622ms real)
  (111556 rewrites/second)
```

We now turn our attention to the PRIORITY example, whose idea was also introduced in Section 4.1. Assume M and N are two sites that return the values 1 and 2, respectively. The two sites are executed concurrently and the result is the response of M if it responds immediately, otherwise the result is the response of either site. In a sense, M is given priority within a time limit after which the two sites are treated equally.

In the absense of delays, PRIORITY will always result in publishing the value 1 of M. This can be checked with the following command (we check whether the result can be that of N before the timeout).

```
Maude> search ['eDelayedN := rtimer(1) > 'vz > (let('vu{0}) where 'vu :in
        'sN()) ; let('vx{0}) where 'vx :in 'sM() | 'eDelayedN()] =>*
        < E:Expr , (tr : e:EventList .
                         (let < 2, H:Handle | N:Nat >) .
                         e':EventList) | R:Record > .
=>* <
    E:Expr,R:Record | tr : e:EventList . (let < 2,H:Handle | N:Nat >) .
    e':EventList > .

No solution.
states: 51  rewrites: 1399 in 10ms cpu (23ms real)
  (139900 rewrites/second)
```

This means that before the timeout is up, the value 2 can never be the value of the expression. On the other hand, the result could be the value of `'sM`, namely 1, as shown by the sample run below.

```
Maude> rew ['eDelayedN := rtimer(1) > 'vz > (let('vu{0}) where 'vu :in
              'sN()) ; let('vx{0}) where 'vx :in 'sM() | 'eDelayedN()] .
rewrites: 503 in 0ms cpu (4ms real) (~ rewrites/second)
```

```
result Conf: < zero,(tr : tau . ('sM < nilA,h(0) | 0 >) . (rtimer < 1,
  h(1) | 0 >) . (h(0) ? 1 | 0) . tau . (let < 1,h(2) | 0 >) . (h(2) ?
  1 | 0) . !! 1 | 0) | (con : 'eDelayedN nilF := rtimer(1) > 'vz > (
  let('vu{0}) where 'vu :in 'sN(nilA))) | (clk : halt) | (msg : [self,
  sig,h(1)]) | hdl : h(3) >
```

9.3 Model Checking Using Maude's LTL Model Checker

Section 9.2 above described a few examples illustrating how the Maude specifications can be used to formally verify some simple safety properties using Maude's breadth-first search capability. However, using Maude's LTL Model Checker, more complex safety as well as liveness properties of finite-state systems can be checked. In this section, some of the model-checking capabilities of Maude are illustrated through an implementation of the Dining Philosophers problem in Orc.

Before we present the model-checking examples below, a couple of minor changes need to be made to the specifications above to directly support model-checking. First, recall that an Orc configuration contains a field `hdl : h(n)` with `n` a natural number to maintain the next handle name available. Since `n` is unbounded, it might grow unnecessarily arbitrarily large (for example, a recursive expression calling some sites repeatedly) while the number of site calls in any expression is always finite. This is easily solved by keeping track of the set of handle names being used in the expression, instead of maintaining the next available handle name. The site call and site return rules are modified accordingly to maintain this set. The other, perhaps more subtle, issue is the use of traces of events in a configuration. The trace of a configuration is allowed to grow indefinitely even for what could otherwise be a finite-state system (the Dining Philosophers is an example of such a system). For $\mathcal{R}^{red}_{Orc}$, this is not a real issue as events can be eliminated altogether (see Section 6). For $\mathcal{R}^{sos}_{Orc}$, however, events cannot just be removed. Instead, what can be done is to disregard the history of events that a configuration has taken[8]. This is achieved by a single equation,

```
eq < f , r | (tr : s:Event) > = < f , r | (tr : nil) > .
```

Therefore, an event is used only to build a proof of a single-step rewrite. Once the step is taken, the event is removed from the trace.

We denote by $\hat{\mathcal{R}}^{sos}_{Orc}$ and $\hat{\mathcal{R}}^{red}_{Orc}$ the modified versions of $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$, respectively. With these technicalities out of the way, we are ready to present the model-checking examples.

The Dining Philosophers Problem and its Specification in Maude

The Dining Philosophers problem (DF) is a classical metaphor in the field of computer science introduced by E. Dijkstra to model a general resource sharing

[8]Note that, by Lemma 2, this change does not affect how a configuration evolves and, thus, the original behavior is preserved.

problem along with possible solutions. The metaphor describes a situation in which n philosophers (representing processes) are sitting on a table having n forks (corresponding to shared resources). A philosopher is either thinking, hungry, or eating. A hungry philosopher will attempt to pick up one of the forks next to him and then the other. If both attempts are successful, the philosopher goes into the eating state, eats, returns the forks one at a time, and then goes into the thinking state, from which he can transition back to the hungry state.

A specification of the dining philosophers problem in Orc is given in [27], and shown below. In a configuration of n philosophers, the ith philosopher is modeled by the Orc expression P_i:

$$\begin{aligned} P_i \quad := \quad & \mathit{fork}[i](\mathit{get}) \gg \mathit{fork}[i'](\mathit{get}) \gg \mathit{eat}[i]() \gg \\ & \mathit{fork}[i](\mathit{put}) \gg \mathit{fork}[i'](\mathit{put}) \gg P_i() \end{aligned}$$

with $i' = i + 1 \mod n$. The ith fork is represented by the site *fork*[i], and the site *eat* models the eating stage of a philosopher.

In Maude, an Orc configuration is first extended with a new field `t-forks : MX` that maintains a set of names `MX` of forks that are currently taken. Then, new site names for *eat* and *fork*[i] sites, and new constants for the messages *get* and *put* are introduced.

```
op eat : -> SiteName [ctor] .
op fork[_] : Nat -> SiteName [ctor] .
ops put get : -> Const [ctor] .
```

When a site *fork*[i] is called with the value *get*, a signal is published granting the philosopher that initiated the call access to the fork, if the fork is not in use.

```
ceq < f , r | (msg : (rho [self, app(fork[i], get, 0), h])) |
      (t-forks : MX) > =
    < f , r | (msg : (rho [self, sig, h])) |
      (t-forks : MX . fork[i]) >
        if not (fork[i] in MX) .
```

If the fork site is called with a *put* message and the fork is currently is in use, a signal is sent back confirming successful release of the fork and the fork name is dropped from the taken forks set.

```
eq < f , r | (msg : (rho [self, app(fork[i], put, 0), h])) |
     (t-forks : MX . fork[i]) > =
   < f , r | (msg : (rho [self, sig, h])) | (t-forks : MX) > .
```

The *eat* site captures the eating state of philosophers. A call to an *eat* site simply publishes a signal back to the caller.

Finally, an operator `df(n)`, with `n` a positive integer, is defined to construct an Orc program of `n` philosophers.

```
op df : NzNat -> Prog .
op df-decl : Nat NzNat -> DeclList .
op df-exp : Nat -> Expr .

ceq df(n) = df-decl(n - 1,n) ; df-exp(n - 1) if n > 1 .
 eq df-exp(s(m)) = phil[s(m)]() | df-exp(m) .
 eq df-exp(0) = phil[0]() .
 eq df-decl(s(m), n) = phil[s(m)] nilF :=
      fork[s(m)](get) > 'vt > fork[(s(m) + 1) rem n](get) > 'vt >
      eat() > 'vt >
      fork[s(m)](put) > 'vt > fork[(s(m) + 1) rem n](put) > 'vt >
      phil[s(m)]() ; df-decl(m, n) .
 eq df-decl(0, n) =  phil[0] nilF :=
      fork[0](get) > 'vt > fork[1](get) > 'vt > eat() > 'vt >
      fork[0](put) > 'vt > fork[1](put) > 'vt > phil[0]() .
```

where `m` is a natural number and `phil[i]` is an expression name for the *i*th philosopher.

LTL Model Checking

Linear Temporal Logic (LTL) is a very widely used property specification language. It is an expressive logic that can be used to specify properties of concurrent systems with infinite behaviors. Terms in the logic are temporal formulae that are constructed out of a set of operators and logical connectives, which include, among others, $\neg$ (negation), $\wedge$ (conjunction), $\Diamond$ (eventually) and $\Box$ (henceforth). The logic has well-established proof methods and decision procedures that, when given a state in a model of the system in question and an LTL formula, can be used to check the satisfiability of the formula at that state. The natural model for a temporal logic specification is a Kripke structure, which is a triple having a set of states, a total transition relation, and a labeling function assigning to each state a set of atomic propositions that hold in that state. Maude's LTL Model Checker builds on the foundation that every rewrite theory in rewriting logic has an underlying Kripke structure. The tool implements a high-performance, explicit-state model checker that allows for the specification and verification of LTL properties of Maude's system modules. For a detailed discussion of the tool and its theory, the reader is referred to [14, 13], and to [11] for a textbook on the subject of model-checking. To simplify analysis and presentation here, we assume no delays and limit the clock ticks to one[9]. We follow a verification strategy similar to the one explained in [22].

A fundamental property of DF is *relative exclusion*, which asserts that no two adjacent philosophers may eat at the same time. To model-check this property, we define the following parameterized *eats* predicate,

```
eq < (eat() > x > f) > x' > phil[i]() | f', r > |= eats(i) = true .
```

[9]timing and delays are not relevant for this example and, thus, by limiting the clock to one tick, no interesting behavior is lost under these assumptions.

The predicate *eats(i)* is true in any state in which the ith philosopher is currently eating, i.e. P_i is ready to call the *eat* site. Next, the relative-exclusion property is given by the operator `rel-excl(n)`, where `n` is the number of philosophers, defined below.

```
op rel-excl : NzNat -> Formula .
op re : Nat NzNat -> Formula .

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)) .
```

Now we show that the the relative exclusion property is satisfied by an instance of DF with four philosophers (given by `df(4)`) by issuing the following command in Maude using $\hat{\mathcal{R}}^{red}_{Orc}$,

```
Maude> red modelCheck( { df(4) } , rel-excl(4) ) .
rewrites: 493754 in 1780ms cpu (2298ms real) (277389 rewrites/second)
result Bool: true
```

where the operator $\{$`P`$\}$ constructs an initial Orc configuration given a program `P`.

However, the program `df(4)` 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 (in both $\hat{\mathcal{R}}^{s}_{Orc}$ and $\hat{\mathcal{R}}^{red}_{Orc}$), a configuration $\mathcal{C}$ being deadlocked coincides with the *eagerEnabled*($\mathcal{C}$) not being true. Therefore, we define the *enabled* predicate accordingly.

```
ceq < f , r > |= enabled = true if eagerEnabled(< f , r >) .
```

The deadlock-freeness property given by `no-deadlock` is defined simply as follows.

```
eq no-deadlock = [] enabled .
```

Now, we can use the model checker 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(4) } , no-deadlock ) .
reduce in DF-CHECK : modelCheck({df(4)}, no-deadlock) .
rewrites: 4598 in 10ms cpu (22ms real) (459800 rewrites/second)
result ModelCheckResult: counterexample({< phil[0](nilA) | phil[1](nilA)
  | phil[2](nilA) | phil[3](nilA),(tr : (nil).EventList) | ...
```

One solution to the deadlock problem is to impose a restriction on the order in which 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(4) } , no-deadlock ) .
rewrites: 22540262 in 83390ms cpu (88047ms real) (270299 rewrites/second)
result Bool: true
```

10 Performance Comparison

In this section we compare the performance of the different variants of the rewriting semantics of Orc presented in previous sections. We emphasize the performance advantage that the reduction semantics enjoys over the SOS-based semantics. In particular, we compare four variants of the rewriting semantics of Orc:

1. The asynchronous real-time SOS-based semantics using the *eagerEnabled* predicate, $\mathcal{R}_{Orc}^{asos}$.
2. The synchronous real-time SOS-based semantics using both the *eagerEnabled* and the *intAction* predicates, $\mathcal{R}_{Orc}^{sos}$.
3. The asynchronous real-time reduction semantics using the notion of eager configurations, $\mathcal{R}_{Orc}^{ared}$.
4. The synchronous real-time reduction semantics using both notions of eager configurations and active expressions, $\mathcal{R}_{Orc}^{red}$.

Throughout all experiments, performance is measured in terms of the time taken to perform a particular task. The numbers reported by Maude are accurate enough for our purposes. The tasks are: (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 problem instances of the dining philosophers problem using Maude's LTL model checker. For the first two tasks, the clock is limited to ten clock ticks, and pseudo-random delays are assumed. However, for the model checking task, time is limited to a single clock tick with no delays (see Section 9.3). The results of these experiments are summerized, respectively, in Table 6, Table 7, and Table 8. To guarantee fairness in comparison, the experiments were carried out on the same machine, using the same version of Maude, and under the same operating conditions[10].

[10]The experiments were carried out on a 3.2GHz dual-core machine with 2GB of memory using Maude 2.3.

	$\mathcal{R}^{asos}_{Orc}$	$\mathcal{R}^{sos}_{Orc}$	$\mathcal{R}^{ared}_{Orc}$	$\mathcal{R}^{red}_{Orc}$
TIMED-MCALL	10	2,178	4	4
TIMEOUT	13	27	2	2
PRIORITY	21	47	3	2
PARALLEL-OR	141	3,247	3	3

Table 6: A performance comparison of different variants of rewriting semantics of Orc using the the `rewrite` command applied to four programs. A number in the table represents the CPU time in milliseconds, as reported by Maude, to simulate a run of an Orc program.

	$\mathcal{R}^{asos}_{Orc}$	$\mathcal{R}^{sos}_{Orc}$	$\mathcal{R}^{ared}_{Orc}$	$\mathcal{R}^{red}_{Orc}$
TIMED-MCALL	325,398	∞	58,703	34,396
TIMEOUT	127	376	32	31
PRIORITY	915	1,921	422	188
PARALLEL-OR	56,922	143,158	10,117	4,861

Table 7: A performance comparison of different variants of rewriting semantics of Orc using the `search` command applied to four programs. A number in the table represents the CPU time in milliseconds, as reported by Maude, to explore all the states of an Orc program.

Problem Size	$\hat{\mathcal{R}}^{asos}_{Orc}$	$\hat{\mathcal{R}}^{sos}_{Orc}$	$\hat{\mathcal{R}}^{ared}_{Orc}$	$\hat{\mathcal{R}}^{red}_{Orc}$
2	109	213	94	53
3	12,702	51,268	4,830	1,502
4	∞	∞	243,927	45,860

Table 8: A performance comparison of different variants of rewriting semantics of Orc using Maude's LTL model checker applied to three instances of the dining philosohpers problem. A number in the table represents the CPU time in milliseconds, as reported by Maude, to model check the absence of deadlock in `df-df(n)`.

In Table 6, it is clear that the reduction semantics of Orc is much more efficiently executable than the SOS-based semantics. This is mainly because the SOS-based semantics makes extensive use of rewrite rules that are mostly conditional with rewrites in their conditions, which are by their non-deterministic nature, more expensive to check than unconditional rules. Furthermore, rules in the SOS-based semantics are always applied at the top (at the level of a configuration) as opposed to being localized to subterms where actual changes of state occur. The reduction semantics, on the other hand, minimizes both the number of rewrite rules and the number of rewrites in the conditions, while using equations to specify the deterministic features of the language, and hence, its efficiency advantage. We also note in Table 6 that the synchronous SOS-based semantics given by $\mathcal{R}_{Orc}^{sos}$, using both the *eagerEnabled* and the *intAction* predicates, is much slower than the asynchronous one using only the *eagerEnabled* predicate. This is due to the fact that in $\mathcal{R}_{Orc}^{sos}$, the rewrite rules for the composition operators had to be duplicated to distinguish the different possible transitions that can be made in order to be able to restrict the site return transition. This increase in the number of conditional rewrite rules, along with need for additional checks for the *intAction* predicate, makes $\mathcal{R}_{Orc}^{sos}$ much less efficient than $\mathcal{R}_{Orc}^{asos}$.

Using the `search` command in Maude, we observe a similar pattern in Table 7 to that in Table 6 with the `rewrite` command. Obviously, the performance gap between the SOS-based semantics and the reduction semantics is larger with the `search` command since searching tries to build proofs of all reachable states, exposing it to the limitations of the SOS-based semantics even more. Furthermore, the performace difference becomes even more pronounced as expressions become more complex. This is justified since the more complex an expression is, the larger the number of compositions used, which translates into a larger number of conditional rewrite checks in the SOS-based semantics. Finally, we note in Table 7 that the synchronous reduction semantics is about twice as efficient as the asynchronous one, which is probably due to the fact that the state space to be explored is often considerably reduced in the synchronous semantics.

With the SOS-based semantics, Maude's LTL model checker successfully reported the result of model checking the dining philosophers specification `df-df(n)` against the deadlock-freeness property for $n = 2, 3$. For any larger problem instance, the model checker would run out of memory. Using the synchronous reduction semantics, the model checker was able model check a problem instance with four philosophers. Any larger instance, however, would cause the model checking to run out of memory. In any model checking experiment, we again observe a similar pattern in which the reduction semantics, especially the synchronouse semantics, is much more efiicient than the SOS-based semantics.

11 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 [27] with n philosophers, there are n subexpressions running in parallel, one for each philosopher. In more practical web-based 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 6 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.

11.1 Distributed Orc Semantics

A *distributed Orc configuration* is modeled by a multiset of objects and messages (constructed by an associative and commutative juxtaposition operator $\_\_$). We describe these objects and messages in detail next.

Objects. An object is a term of the form $\langle \mathit{Oid} : \mathit{Cid} \mid \mathit{AttributeSet} \rangle$, with *Oid* the object identifier, *Cid* the class of the object, and *AttributeSet* a set of attributes, which are typically of the form *AttributeName* : *AttributeValue*. In a distributed Orc configuration, there are three classes of objects, namely, *expression*, *site*, and *clock* objects:

1. An *expression* object is an object of class *Expr* having three attributes: (i) *exp*, which holds an Orc expression to be computed; (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. For example, the object representing the program TIMEOUT is

$$\begin{aligned}&\langle o_1 : \mathit{Expr} \mid \mathit{exp} : f(3),\\ &\mathit{con} : f(t) =_{def} \mathit{let}(z) \ \textbf{where}\ z :\in M \mid \mathit{rtimer}(t) > x > \mathit{let}(0),\ \mathit{hdl} : \emptyset\rangle\end{aligned}$$

2. A *site* object is an object of class *Site* with the following attributes:
   (a) *name*: the name of the site, such as *if*, *rtimer*, *CNN*, *BBC*, ...etc.
   (b) *op*: the current operation being performed by the site, which can either be $exec(C, h, Oid)$ or *free*. 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.
   (c) *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.

   For instance, the fundamental site *if* is represented by the object $\langle o_{if} : Site \mid name : if, op : free, state : nil\rangle$, while a more sophisticated site, say *CNN*, could be represented by the following object

   $$\langle o_{CNN} : Site \mid name : CNN, op : free, state : (d_1, p_1), \ldots, (d_n, p_n)\rangle$$

   where a pair (d_i, p_i) maintains the news page p_i for the date d_i.

3. A *clock* object is a simple object of class *Clock*, which maintains a single field, called *clk*, representing the current clock time.

   $$\langle o_{clock} : Clock \mid clk : c_m\rangle$$

Messages. A *message* is either a *site call* message of the form $M \leftarrow sc(Oid, C, h, m)$, with M the name of the site being called, and *Oid* the object identifier of the caller expression object, or a *site return* message of the form $Oid \leftarrow sr(c, h, m)$, with *Oid* 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 6[11], 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. For instance, a site call is modeled by the following two rules

$$\begin{array}{lcl} \text{SITECALL} & : & \langle Oid : Expr \mid exp : f, AS\rangle \\ & & \rightarrow \langle Oid : Expr \mid exp : sc^{\uparrow}(f', M, C), AS\rangle \text{ if } f \rightarrow sc^{\uparrow}(f', M, C) \\ \text{SITECALL*} & : & M(C) \rightarrow sc^{\uparrow}(\gamma, M, C) \end{array}$$

The first rule resembles the SiteCall of the reduction semantics except that it is defined on an Orc expression object, while the second rule is exactly the

[11]We use the variant of the reduction semantics in which traces are not used.

same as the corresponding rule in the reduction semantics. The operator $sc^{\uparrow}$ is also very similarly defined except when it reaches the top of the expression, where the effect of a site call is reflected:

$$\begin{aligned}&\langle Oid : Expr \mid exp : sc^{\uparrow}(f', M, C), hdl : H, AS\rangle \\ &\qquad = \langle Oid : Expr \mid exp : sc^{\downarrow}(f, h), hdl : h\#H, AS\rangle \\ &\qquad\qquad M \leftarrow sc(Oid, C, h, 0) \text{ if } h := gFresh(H)\end{aligned}$$

The operator $sc^{\downarrow}$, which propagates the new handle down the expression tree to the appropriate subexpression, is defined exactly as before. Notice here the site call caused a message to be emitted into the distributed configuration so that a site object with the name M, if defined, may consume and respond to it. The changes to the specifications of the other internal and external actions have the same pattern.

Finally, the timed, synchronous semantics of Orc is also achieved using the same "active expression" and "eager configuration" notions, with the exception that now the *tick* rule encloses the entire distributed configuration into a system, so that the global effect of a clock tick is properly captured:

$$\begin{aligned}&\{\langle o_{clock} : Clock \mid clk : c_m\rangle\ CF\} \to \{\langle o_{clock} : Clock \mid clk : c_{m+1}\rangle\ \delta(CF)\} \\ &\qquad\qquad \text{if } eager(\langle o_{clock} : Clock \mid clk : c_m\rangle\ CF) \neq true\end{aligned}$$

The complete specification of the distributed semantics in Maude can be found in Appendix A.2. Next, we illustrate through an application how Maude's LTL model checker can be used to verify properties of distributed Orc systems.

11.2 Case Studies

We present two distributed applications: (1) AUCTION, an online auction management program; and (2) MEETING, a meeting scheduler and monitor program, both of which were inspired by the examples given in [27]. We illustrate how the distributed semantics facilitates the formal verification of programs using Maude's LTL model checker.

11.2.1 Managing an Online Auction

The distributed Orc auction program uses a number of expression declarations that we describe first. 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[12].

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

[12]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*.

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 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.

$$\begin{aligned} &\mathit{Bidding}(B) =_{def} \mathit{auction}(\mathit{getNext}) > \mathit{item} > \mathit{Bids}(\mathit{item}_0, \mathit{item}_1, \mathit{item}_2, B, 0) > w > \\ &\qquad (\mathit{if}(w_1 = 0) \gg \mathit{Bidding}(B) \mid \mathit{if}(w_1 \neq 0) \gg w_1(\mathit{won}, \mathit{item}_0, w_0)) \end{aligned}$$

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

$$\begin{aligned} &\mathit{Bids}(id, \mathit{duration}, \mathit{bid}, B, \mathit{winner}) =_{def} \mathit{if}(\mathit{duration} = 0) \gg \mathit{let}(\mathit{bid}, \mathit{winner}) \\ &\quad \mid \mathit{if}(\mathit{duration} \neq 0) \gg \mathit{Collect}(\mathit{nextBid}, B, id, \mathit{bid}) > \mathit{bidList} > \\ &\qquad \mathit{MaxBid}(\mathit{bidList}) > m > \mathit{rtimer}(1) \gg \mathit{Bids}(id, \mathit{duration} - 1, m_0, B, m_1) \end{aligned}$$

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 $(\mathit{bid}, \mathit{bidder})$.

$$\begin{aligned} &\mathit{Collect}(m, B, id, \mathit{minBid}) =_{def} \mathit{if}(\mathit{empty}(B)) \gg \mathit{let}(\mathit{nil}) \mid \mathit{if}(\neg \mathit{empty}(B)) \gg \\ &\quad (\mathit{append}(x, xs) \ \mathbf{where}\ x :\in B_0(m, id, \mathit{minBid}) \\ &\qquad\qquad \mathbf{where}\ xs :\in \mathit{Collect}(m, \mathit{tail}(B), id, \mathit{minBid})) \end{aligned}$$

In addition to the clock object, the initial configuration of AUCTION consists of the followig objects:

- Two expression objects: the *posting* expression object and the *bidding* expression object,

$$\langle o_1 : \mathit{Expr} \mid \mathit{exp} : \mathit{Posting}(\mathit{seller}_0),\ \mathit{con} : \mathit{PostingDecl},\ \mathit{hdl} : \emptyset\rangle$$
$$\langle o_2 : \mathit{Expr} \mid \mathit{exp} : \mathit{Bidding}(b_0, b_1, b_2),\ \mathit{con} : \mathit{BiddingDecl}, \mathit{BidsDecl}, \mathit{CollectDecl},\ \mathit{hdl} : \emptyset\rangle,$$

- The fundamental site objects:

$$\langle o_{if} : \mathit{Site} \mid \mathit{name} : \mathit{if}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil}\rangle$$
$$\langle o_{let} : \mathit{Site} \mid \mathit{name} : \mathit{let}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil}\rangle$$
$$\langle o_{rtimer} : \mathit{Site} \mid \mathit{name} : \mathit{rtimer}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil}\rangle$$
$$\langle o_{atimer} : \mathit{Site} \mid \mathit{name} : \mathit{atimer}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil}\rangle$$
$$\langle o_{clock} : \mathit{Site} \mid \mathit{name} : \mathit{clock}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil}\rangle$$
$$\langle o_{signal} : \mathit{Site} \mid \mathit{name} : \mathit{signal}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil}\rangle$$

- A seller site object, whose state maintains two items to be auctioned

$$\langle o_{seller} : \mathit{Site} \mid \mathit{name} : \mathit{seller}_0, \mathit{op} : \mathit{free}, \mathit{state} : [(t_1, 5, 500)), (t_2, 8, 700)]\rangle$$

- Three bidder site objects named b_i, for $i = 0, 1, 2$,

  $$\langle o_{b_i} : \mathit{Site} \mid \mathit{name} : b_i, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil} \rangle$$

  The bidder objects use their state during program execution to keep track of their bids and the items they win.

- An auction site object that manages auction items and services bidder requests

  $$\langle o_{auction} : \mathit{Site} \mid \mathit{name} : \mathit{auction}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil} \rangle$$

  The state field is used during program execution to maintain a list of available items and another of bidding requests.

- A site object for the *MaxBid* site

  $$\langle o_{MaxBid} : \mathit{Site} \mid \mathit{name} : \mathit{MaxBid}, \mathit{op} : \mathit{free}, \mathit{state} : \mathit{nil} \rangle$$

  The site is stateless as it merely computes the highest bid amongst a list of bids.

The behaviors of the different site objects are specified easily in Maude. We refer the reader to Appendix A.2.4 for the complete specification.

Using this specification in Maude, 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:

1. $\mathit{hasbid}(t)$, which is true in a state where the item t has been bid on,
2. $\mathit{sold}(t)$, which is true in a state where the item t has been sold,
3. $\mathit{max}(t)$, which is true in a state where the item t has been sold to the highest bidder,
4. $\mathit{conflict}(t)$ which is true whenever t has two or more winning bidders.

The correctness properties along with Maude's output are as follows,

1. An item that has at least one bid is eventually sold: $\Box \bigwedge_i (\mathit{hasbid}(t_i) \rightarrow \Diamond \mathit{sold}(t_i))$. In Maude, the operator `commitAll` specifies this property for the items t_1 and t_2.

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

2. An item is always sold at the maximum bid to the highest bidder: $\Box \bigwedge_i (\mathit{sold}(t_i) \rightarrow \mathit{max}(t_i))$. In Maude, the operator `winAll` specifies this property for the items t_1 and t_2.

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

3. An item cannot have two winners: $\neg\Diamond \bigvee_i \mathit{conflict}(t_i)$. This is specified for the items t_1 and t_2 in Maude using the operator `uniqueWinnerAll`.

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

11.2.2 Scheduling a Meeting

The meeting scheduler program MEETING manages the process of arranging a meeting time and location among a group of participants, and monitors the resulting room reservation against possible preemptions. The top level expression declaration *MeetingDecl* used by the program is shown below.

```
$MeetingScheduler(P) =_{def}$
    $if(empty(T)) \gg let(false)$
    $\mid if(\neg empty(T)) \gg ArrangeMeeting(P)$
```

where P is a list of participants. $MeetingScheduler(P)$ publishes the value *true* if it successfully schedules a meeting venue. Otherwise, the value *false* is published. The expression $ArrangeMeeting(P)$, whose declaration *ArrangeDecl* is shown below, first solicits schedules of participants P. If enough responses are received, it then uses a site *MeetTime* to compute the earliest possible time t for the meeting. If successful, the site *RoomReserve* is called with the meeting time t and the number of participants n to reserve a suitable room r for the meeting. If any of these steps fail, a *cancel* message is broadcast to all participants.

```
$ArrangeMeeting(P) =_{def} GetSchedules(P) > s >$
    $(if(size(s) < 2) \gg Broadcast(cancel, P) \gg let(false)$
    $\mid if(size(s) \geq 2) \gg MeetTime(s) > t >$
        $(if(t = 0) \gg Broadcast(cancel, P) \gg let(false)$
        $\mid if(t \neq 0) \gg RoomReserve(t, size(s)) > r >$
            $(if(r = 0) \gg Broadcast(cancel, P) \gg let(false)$
            $\mid if(r \neq 0) \gg Broadcast(t, r, P) \gg Monitor(t-1, r, size(s), P)$
            $)$
        $)$
    $)$
```

Once a reservation is made, the expression *Monitor* is called to monitor the status of the reservation using the site *RoomCanceled*. If the reservation is preempted before the threshold, a new reservation is attempted.

```
$Monitor(t;r;n;P) =_{def}$ $let(b)$ **where** $b :\in$
    $(atimer(t) \gg let(true)$ |
        $(RoomCanceled(r) \gg RoomReserve(t) > s >$
            $(if(s = 0) \gg Broadcast(cancel, P) \gg let(false)$
            | $if(s \neq 0) \gg let(true)$
            )
        )
    )
```

The declarations for the expressions *GetSchedules*(P) and *Broadcast*(x, P), respectively named *GetSchedulesDecl* and *BroadcastDecl*, are shown below.

```
$GetSchedules(P) =_{def}$
    $if(empty(P)) \gg let(nil)$
    | $(if(\neg empty(P)) \gg append(x, xs)$
            **where** $x :\in ((head(P))(get) \mid rtimer(5))$
            **where** $xs :\in GetSchedules(tail(P))$
    )
```

```
$Broadcast(x;P) =_{def}$
    $if(empty(P)) \gg let(signal)$
    | $if(\neg empty(P)) \gg (head(P))(x) \gg Broadcast(x, tail(P))$
```

The expression *GetSchedules* requests user schedules and waits for responses within a timeout period of five time units.

The Orc program MEETING assumes three participants p_1, p_2 and p_3. It contains one expression object that manages scheduling a meeting among the three participants:

$\langle o_{\text{MEETING}} : Expr \mid exp : MeetingScheduler([p_1, p_2, p_3]),$
$con : MeetingDecl; ArrangeDecl; GetSchedulesDecl; MonitorDecl; BroadcastDecl,$
$hdl : \emptyset\rangle$

In addition to the clock object and the fundamental site objects, MEETING also contains the following site objects:

1. Three participant site objects, each representing a single participating user, of the following form:

   $$\langle s_{p_i} : Site \mid name : p_i, op : free, state : (s, x) >$$

   where the state of a participant is a pair (s, x), with s a schedule and x a status flag indicating whether the user is ready to take part in the meeting to be scheduled, waiting for a response from the scheduler, or has his meeting successfully scheduled or canceled.

2. A *MeetTime* site object of the form,

   $$\langle s_{MeetTime} : Site \mid name : MeetTime, op : free, state : nil >$$

   The site is stateless since it represents a function that computes the earliest meeting time possible given a list of user schedules.

3. A *RoomReserve* site object of the form,

   $$\langle s_{RoomReserve} : Site \mid name : RoomReserve, op : free, state : R >$$

   where R is a database of room availability tuples of the form (r, n, s), with r the room number, n the capacity of the room, and s a list of time slots during which the room r is available.

4. A *RoomCanceled* site object of the form,

   $$\langle s_{RoomCanceled} : Site \mid name : RoomCanceled, op : free, state : M >$$

   where M maintains a list of room reservations to monitor.

We now specify and then verify two correctness properties about MEETING using Maude's LTL model checker. The first property states that if a user does not respond to the scheduler within the timeout (five time units), the user will not ever particpate in the meeting. This can be specified as

$$\Box(timedout \wedge \neg success(n) \rightarrow \neg\Diamond success(n)) \tag{4}$$

where *timedout* is an atomic predicate which is true in a state in which the timeout period has elapsed, and *success*(n) is a state formula which evaluates to true whenever all the n users are successfully scheduled. In Maude, the program, whose initial state is given by `init(3)`, is model checked against this property as follows.

```
Maude> red modelCheck(init(3), scheduled-after-timeout(3)) .
rewrites: 540704307 in 1079855ms cpu (1079848ms real) (500718 rewrites/second)
result Bool: true
```

The operator `scheduled-after-timeout(n)` captures the property given by the formula 4 above. The second property states that every participating user will eventually be informed whether the meeting is successfully scheduled or canceled. This is specified by the formula,

$$\Box \bigwedge_i (participant(i) \rightarrow \Diamond(scheduled(i) \vee canceled(i)))$$

where *participant*(i), *scheduled*(i) and *canceled*(i) are atomic propositions that are true in a state whenever the ith participant is, respectively, waiting for a response from the scheduler, has his meeting successfully scheduled, or has it canceled. The property is model checked in Maude using the following command:

```
Maude> red modelCheck(init(3), [] gr-all(3)) .
reduce in MT-CHECK : modelCheck(init(3), []gr-all(3)) .
rewrites: 541347035 in 1199067ms cpu (1214902ms real) (451473 rewrites/second)
result Bool: true
```

12 Concluding Remarks

We have presented two approaches to the operational semantics of Orc using rewriting logic. The first uses a semantics-preserving transformation from Orc's SOS specification to an SOS-like rewrite theory. The second is in the style of reduction semantics and is based on the distinction between rules and equations in rewriting logic. Both rewrite theories were further refined to support Orc's timed synchronous semantics using two alternative methods: strategy expressions and equationally defined predicates. We have shown that the two refined theories are semantically equivalent through a strong bisimulation, and that the reduction semantics is much more efficiently executable and analyzable using various experiments in Maude. We have also extended 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 [8] to develop such a distributed implementation.

References

[1] J. Baeten and C. Middelburg. Process algebra with timing: Real time and discrete time, 2000.

[2] Jos C. M. Baeten and Jan A. Bergstra. Real time process algebra. *Formal Asp. Comput.*, 3(2):142–188, 1991.

[3] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[4] Christiano Braga and José Meseguer. Modular rewriting semantics in practice. *Electr. Notes Theor. Comput. Sci.*, 117:393–416, 2005.

[5] Roberto Bruni, Hernán Melgratti, and Emilio Tuosto. Translating Orc features into petri nets and the join calculus. *Web Services and Formal Methods*, pages 123–137, 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] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework.* To be published by Springer, 2007.

[9] 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`.

[10] William R. Cook and Jayadev Misra. A structured orchestration language. July 2005. `http://www.cs.utexas.edu/users/wcook/Drafts/OrcCookMisra05.pdf`.

[11] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[12] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. In *Proceedings of STRATEGIES'06*, August 2006.

[13] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc. 10^{th} Intl. SPIN Workshop*, volume 2648, pages 230–234. Springer LNCS, 2003.

[14] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:143–168, April 2004.

[15] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput.*, 117(2):221–239, 1995.

[16] Tony Hoare, Galen Menzel, and Jayadev Misra. A tree semantics of an orchestration language. August 2004. Also available at `http://www.cs.utexas.edu/users/psp/Semantics.Orc.pdf`.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] José Meseguer. Lecture notes on program verification. CS 476, University of Illinois, `http://www-courses.cs.uiuc.edu/~cs476/`, Spring 2005.

[23] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In *AMAST*, pages 364–378, 2004.

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

[25] 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.

[26] 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.

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

[28] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

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

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

[31] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude 2.1. *Electr. Notes Theor. Comput. Sci.*, 117:285–314, 2005.

[32] 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.

[33] 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.

[34] Mark-Oliver Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. In Kokichi 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`.

[35] 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`).

A Maude Specifications

A.1 The SOS-based and the Reduction Semantics

A.1.1 Syntax

```
fmod NAMES is
  protecting QID .

  sorts SiteName ExprName Var IVar Const PreConst ConstList .
  subsorts SiteName ExprName Var < Qid .
  subsorts Nat < Const < PreConst .
  subsorts Const < ConstList .

  op tup : ConstList -> Const [ctor] .
  op tr : Bool -> Const [ctor] .
  op sig : -> Const [ctor] .

  op self : -> SiteName [ctor] .
  op _{_} : Var Nat -> IVar [ctor prec 1] .

  var Q : Qid .

  cmb Q : SiteName if substr(string(Q), 0, 1) == "s" .
  cmb Q : ExprName if substr(string(Q), 0, 1) == "e" .
  cmb Q : Var if substr(string(Q), 0, 1) == "v" .
endfm

fmod PARAMETER is
  protecting NAMES .
  sorts AParam FParam AParamList FParamList .
  subsorts Var < FParam < FParamList .
  subsorts IVar SiteName Const < AParam < AParamList .
  subsorts ConstList < AParamList .

  op nilF : -> FParamList [ctor] .
  op _;_ : FParamList FParamList -> FParamList [ctor assoc id: nilF prec 8] .
  op nilA : -> ConstList [ctor] .
  op _,_ : AParamList AParamList -> AParamList [ctor assoc id: nilA prec 8] .
  op _,_ : ConstList ConstList -> ConstList [ctor assoc id: nilA prec 8] .

endfm

fmod ORC-SYNTAX is
  protecting PARAMETER .
  sorts Prog Decl DeclList Expr Handle .
  subsort Decl < DeclList .

  op h : Nat -> Handle [ctor] .

  op _;_ : DeclList Expr -> Prog [ctor prec 50] .

  op nilD : -> DeclList [ctor] .
  op _;_ : DeclList DeclList -> DeclList [ctor assoc id: nilD prec 40] .
  op __:=_ : ExprName FParamList Expr -> Decl [ctor frozen(3) prec 30] .

  op zero : -> Expr [ctor] .
  op _(_) : SiteName AParamList -> Expr [ctor prec 10] .
  op _(_) : ExprName AParamList -> Expr [ctor prec 10] .
  op !_ : IVar -> Expr [ctor prec 5] .
  op !_ : Const -> Expr [ctor prec 5] .
  op _>_>_ : Expr Var Expr -> Expr [ctor frozen (3) prec 15 gather (e & E)] .
  op _|_ : Expr Expr -> Expr [ctor assoc comm prec 20] .
  op _where_:in_ : Expr Var Expr -> Expr [ctor prec 25 gather (E & e)] .

  op ?_ : Handle -> Expr [ctor prec 1] .

  var x : Var . var f : Expr .
  var E : ExprName . var M : SiteName .
  eq zero > x > f = zero .
  eq zero | f = f .

  *** Syntactic Sugar
  op _:=_ : ExprName Expr -> Decl [prec 30] .
  eq E := f = E nilF := f .

  op _() : SiteName -> Expr [prec 10] .
  op _() : ExprName -> Expr [prec 10] .
  eq M() = M(nilA) .
  eq E() = E(nilA) .

endfm
```

A.1.2 CINNI Substitution Calculus

```
fmod CINNI is
  protecting PARAMETER .
  sort Subst .

  op [_:=_] : Var AParam  -> Subst [ctor] .
  op [shiftup_] : Var -> Subst [ctor] .
  op [lift__] : Var Subst -> Subst [ctor] .
  op __ : Subst IVar -> IVar [ctor] .

  vars n : Nat .
  vars a b : Var .
  vars x : AParam .
  vars S : Subst .

  eq [a := x] a{0} = x .
  eq [a := x] a{s(n)} = a{n} .
 ceq [a := x] b{n} = b{n} if a =/= b .

  eq [shiftup a] a{n} = a{s(n)} .
 ceq [shiftup a] b{n} = b{n}  if a =/= b .

  eq [lift a S] a{0} = a{0} .
  eq [lift a S] a{s(n)} = [shiftup a] S a{n} .
 ceq [lift a S] b{n} = [shiftup a] S b{n} if a =/= b .
endfm

fmod SUBSTITUTION is
  protecting CINNI .
  protecting ORC-SYNTAX .

  op __ : Subst Expr -> Expr [frozen (2)] .
  op __ : Subst AParamList -> AParamList .

  op [_<-_]_ : FParamList AParamList Expr -> Expr [frozen (3)] .

  vars id : Qid . vars n : Nat .
  vars p : AParam . vars P : AParamList .
  vars Q : FParamList .
  vars d : Decl . vars D : DeclList .
  vars ix : IVar . vars x : Var . vars c : Const .
  vars f f' : Expr .
  vars S : Subst .
  vars E : ExprName . vars M : SiteName .
  var h : Handle .

  eq [x ; Q <- p, P] f = [x := p] ([Q <- P] f) .
  eq [nilF <- nilA] f = f .

  eq S zero = zero .
  eq S (id ( P )) = id ( S P ) .
  eq S ! ix = ! (S ix) .
  eq S ! c = ! c .
  eq S (f > x > f') = (S f) > x > ([lift x S] f') .
  eq S (f | f') = (S f) | (S f') .
  eq S (f where x :in f') = ([lift x S] f) where x :in (S f') .
  eq S (? h) = ? h .

  eq S (nilA) = nilA .
 ceq S (x{n}, P) = (S x{n}) , (S P) if P =/= nilA .
  eq S (p , P) = p , (S P) [owise] .

endfm
```

A.1.3 Semantic Infrastructure

```
fmod CLOCK is
  pr NAT .
  sort Clock .
  op halt : -> Clock [ctor] .
  op clock : Nat -> Clock [ctor] .
  eq clock(1) = halt . --- finite state clock
endfm

fmod EVENT is
  protecting ORC-SYNTAX .
  sort Event .
  op _<_,_|_> : SiteName ConstList Handle Nat -> Event [ctor] .
  op _?_|_ : Handle Const Nat -> Event [ctor] .
  op !!_|_ : Const Nat -> Event [ctor] .
  op tau : -> Event [ctor] .
endfm

fmod EVENT-LIST is
  protecting EVENT .
  sort EventList .
  subsort Event < EventList .
  op nil : -> EventList [ctor] .
  op _._ : EventList EventList -> EventList [ctor assoc id: nil] .
endfm

fmod CONTEXT is
  protecting ORC-SYNTAX .
  sort Context .
  subsort Decl < Context .

  op mt : -> Context [ctor] .
  op _,_ : Context Context -> Context [ctor assoc comm id: mt prec 42] .
  op _<-_ : Context Decl -> Context [ctor prec 45 gather (E e)] .

  var E : ExprName .
  var f f' : Expr .
  var sigma : Context .
  var d : Decl .
  var Q Q' : FParamList .

  eq E Q := f , sigma <- E Q' := f' = E Q' := f' , sigma .
  eq sigma <- d = d , sigma [owise] .
endfm

fmod MESSAGES is
  protecting ORC-SYNTAX .
  sorts Msg MsgPool .
  subsort Msg < MsgPool .

  op [_,_,_] : SiteName [ConstList] Handle -> [Msg] [ctor] .
  op empty : -> MsgPool [ctor] .
  op __ : MsgPool MsgPool -> MsgPool [ctor assoc comm id: empty] .

  var M : SiteName . var C : ConstList . var h : Handle .
  var pc : PreConst .
  mb [self, pc, h] : Msg .
  mb [M, C, h] : Msg .
endfm

fmod HANDLE-SET is
  protecting ORC-SYNTAX .
  sort HandleSet .
  subsort Handle < HandleSet .

  op mth : -> HandleSet [ctor] .
  op _#_ : HandleSet HandleSet -> HandleSet [ctor assoc comm id: mth] .

  vars h h' : Handle .
  vars H : HandleSet .
  var n : Nat .

  eq h # h # H = h # H .

  op _usedin_ : Handle HandleSet -> Bool .
  eq h usedin (h' # H) = h == h' or h usedin H .
  eq h usedin mth = false .

  op gFresh : HandleSet -> Handle .
  op gFreshAux : HandleSet Handle -> Handle .

  eq gFresh(H) = gFreshAux(H, h(0)) .
  eq gFreshAux(H, h(n)) = if h(n) usedin H
                          then gFreshAux(H, h(s n))
                          else h(n) fi .
endfm

fmod RECORD-HS is
```

```
  pr EVENT-LIST .
  pr CONTEXT .
  pr CLOCK .
  pr MESSAGES .
  pr HANDLE-SET .

  sorts Index Field Record .
  subsort Field < Record .

  ops tr con clk msg hdl : -> Index [ctor] .

  op null : -> Record [ctor] .
  op _|_ : Record Record -> Record [ctor assoc comm id: null] .
  op _:_ : Index EventList -> Field [ctor] .
  op _:_ : Index Context -> Field [ctor] .
  op _:_ : Index Clock -> Field [ctor] .
  op _:_ : Index MsgPool -> Field [ctor] .
  op _:_ : Index HandleSet -> Field [ctor] .

endfm

fmod RECORD-HC is
  pr EVENT-LIST .
  pr CONTEXT .
  pr CLOCK .
  pr MESSAGES .

  sorts Index Field Record .
  subsort Field < Record .

  ops tr con clk msg hdl : -> Index [ctor] .

  op null : -> Record .
  op _|_ : Record Record -> Record [ctor assoc comm id: null] .
  op _:_ : Index EventList -> Field [ctor] .
  op _:_ : Index Context -> Field [ctor] .
  op _:_ : Index Clock -> Field [ctor] .
  op _:_ : Index MsgPool -> Field [ctor] .
  op _:_ : Index Handle -> Field [ctor] .
endfm

mod RCONF-HC is
 inc RECORD-HC .
 sort Conf .
 op <_,_> : Expr Record -> Conf [ctor frozen(1)] .
endm

mod SOS-RCONF-HC is
 inc RCONF-HC .
 op {_,_} : [Expr] [Record] -> [Conf] [ctor] .
 op [_,_] : [Expr] [Record] -> [Conf] [ctor] .

 vars f f' : Expr .  vars r r' : Record .

 crl [step] : < f , r > => < f' , r' > if {f,r} => [f',r'] .
endm

mod RCONF-HS is
 inc RECORD-HS .
 sort Conf .
 op <_,_> : Expr Record -> Conf [ctor frozen(1)] .
endm

mod SOS-RCONF-HS is
 inc RCONF-HS .
 op {_,_} : [Expr] [Record] -> [Conf] [ctor] .
 op [_,_] : [Expr] [Record] -> [Conf] [ctor] .

 vars f f' : Expr .  vars r r' : Record .

 crl [step] : < f , r > => < f' , r' > if {f,r} => [f',r'] .
endm

mod SIMCOUNTER is
  pr NAT .
  op counter : -> [Nat] .
  rl [count] : counter => s(counter) .
  rl [eval]  : counter => 0 .
  eq s_^5(counter) = counter .  --- finite state counter
endm

mod SIMRANDOM is
  pr RANDOM .
  pr SIMCOUNTER .
  pr RAT .
  pr CONVERSION .

  op rand : -> [Nat] .

  ----rl [rnd] : rand => floor((random(counter) / 4294967296) * 10) .
  eq rand = 0 .
endm
```

A.1.4 Asynchronous SOS-Based Semantics

```
mod ORC-SEMANTICS is
  inc SOS-RCONF-HC .
  pr SUBSTITUTION .
  pr SIMRANDOM .
```

```
 op initCon : DeclList -> Context .
 op [_] : Prog -> Conf .
 op app : SiteName ConstList Nat -> PreConst .
 op eagerEnabled : Conf -> [Bool] [frozen] .

 var d : Decl . var D : DeclList .
 var h : Handle .
 var M : SiteName . var c : Const . var x : Var .
 var P : AParamList . var C : ConstList .
 vars f f' g g' : Expr .
 var s : Event . var n m : Nat .
 var E : ExprName . var Q : FParamList .
 var t : EventList .
 var sigma : Context .
 var rho : MsgPool .
 vars r r' : Record .

 eq initCon(nilD) = mt .
 eq initCon(D ; d) = initCon(D) <- d .

 eq eagerEnabled(< f, r >) = eagerEnabled({f, r}) .

 eq [D ; f] = < f , (tr : nil) | (con : initCon(D)) |
                   (clk : clock(0)) | (msg : empty) | (hdl : h(0)) > .


 rl [SiteCall] : { M(C) , (tr : t) | (msg : rho) |
                         (hdl : h(n)) | (clk : clock(m)) | r}
      => [ ? h(n) , (tr : (t . M < C , h(n) | m >)) |
                       (msg : (rho [M, C, h(n)])) |
      (hdl : h(s n)) | (clk : clock(m)) | r] .
 eq eagerEnabled({ M(C) , (tr : t) | (msg : rho) | (hdl : h(n)) |
                            (clk : clock(m)) | r}) = true .


ceq [M, C, h] = [self, app(M, C, rand), h] if M =/= self .


 rl [SiteRet] : { ? h , (tr : t) | (msg : (rho [self, c, h])) |
                         (clk : clock(m)) | r}
             => [ ! c , (tr : (t . h ? c | m)) | (msg : rho) |
                         (clk : clock(m)) | r] .
 eq eagerEnabled({ ? h , (tr : t) | (msg : (rho [self, c, h])) |
                                       (clk : clock(m)) | r}) = true .


 rl [Pub] : {! c , (tr : t) | (clk : clock(m)) | r}
            => [zero, (tr : t . (!! c | m)) | (clk : clock(m)) | r] .
 eq eagerEnabled({! c , (tr : t) | (clk : clock(m)) | r}) = true .
 rl [Def] : { E(P), (tr : t) | (con : (sigma , E Q := f)) | r }
           => [ ([Q <- P] f), (tr : t . tau) |
                                    (con : (sigma , E Q := f)) | r ] .
 eq eagerEnabled({ E(P), (tr : t) | (con : (sigma , E Q := f)) | r })
    = true .


 crl [Sym] : { f | g , (tr : t) | r} => [ f' | g, (tr : (t . s)) | r']
    if { f , (tr : nil) | r} => [ f' , (tr : s) | r' ] .
 ceq eagerEnabled({ f | g , (tr : t) | r}) = true
                     if eagerEnabled({ f , (tr : nil) | r}) .


 crl [Seq1V] : { f > x > g , (tr : t) | r}
               => [ (f' > x > g) | ([x := c] g) , (tr : t . tau) | r' ]
if { f , (tr : nil) | r }
                                 => [ f' , (tr : (!! c | m)) | r' ] .
 ceq eagerEnabled({ f > x > g , (tr : t) | r}) = true
                           if eagerEnabled({ f , (tr : nil) | r }) .

 crl [Seq1N1] : { f > x > g , (tr : t) | r}
                              => [ f' > x > g , (tr : t . tau) | r' ]
 if { f , (tr : nil) | r } => [ f' , (tr : tau) | r' ] .

 crl [Seq1N2] : { f > x > g , (tr : t) | r}
                      => [ f' > x > g , (tr : (t . h ? c | m)) | r' ]
if { f , (tr : nil) | r }
                                => [ f' , (tr : (h ? c | m)) | r' ] .

 crl [Seq1N3] : { f > x > g , (tr : t) | r}
                  => [ f' > x > g , (tr : (t . M < C , h | m >)) | r' ]
if { f , (tr : nil) | r }
                  => [ f' , (tr : (M < C , h | m >)) | r' ] .


 crl [Asym1V] : { g where x :in f , (tr : t) | r}
                             => [ ([x := c] g) , (tr : t . tau) | r' ]
if { f , (tr : nil) | r }
                                 => [ f' , (tr : (!! c | m)) | r' ] .
 ceq eagerEnabled({ g where x :in f , (tr : t) | r}) = true
                           if eagerEnabled({ f , (tr : nil) | r })  .


 crl [Asym1N1] : { g where x :in f , (tr : t) | r}
                         => [ g where x :in f' , (tr : t . tau) | r' ]
                 if { f , (tr : nil) | r } => [ f' , (tr : tau) | r' ] .

 crl [Asym1N2] : { g where x :in f , (tr : t) | r}
                  => [ g where x :in f' , (tr : (t . h ? c | m)) | r' ]
```

```
if { f , (tr : nil) | r }
                                  => [ f' , (tr : (h ? c | m)) | r' ] .

 crl [Asym1N3] : { g where x :in f , (tr : t) | r}
            => [ g where x :in f' , (tr : (t . M < C , h | m >)) | r' ]
if { f , (tr : nil) | r }
                            => [ f' , (tr : (M < C , h | m >)) | r' ] .


 crl [Asym2] : { g where x :in f , (tr : t) | r}
                         => [ g' where x :in f , (tr : (t . s)) | r' ]
   if { g , (tr : nil) | r } => [ g' , (tr : s) | r' ] .
 ceq eagerEnabled({ g where x :in f , (tr : t) | r}) = true
                              if eagerEnabled({ g , (tr : nil) | r }) .

endm
```

A.1.5 Synchronous SOS-Based Semantics

```
mod ORC-SEMANTICS is
  inc SOS-RCONF-HC .
  pr SUBSTITUTION .
  pr SIMRANDOM .

  op initCon : DeclList -> Context .
  op [_] : Prog -> Conf .
  op app : SiteName ConstList Nat -> PreConst .
  op eagerEnabled : Conf -> [Bool] [frozen] .
  op intActions : Conf -> [Bool] [frozen] .

  var d : Decl . var D : DeclList .
  var h : Handle .
  var M : SiteName . var c : Const . var x : Var .
  var P : AParamList . var C : ConstList .
  vars f f' g g' : Expr .
  var s : Event . var N T : Nat .
  var E : ExprName . var Q : FParamList .
  var t : EventList .
  var sigma : Context .
  var rho : MsgPool .
  vars r r' : Record .

  eq initCon(nilD) = mt .
  eq initCon(D ; d) = initCon(D) <- d .
  eq [D ; f] = < f , (tr : nil) | (con : initCon(D)) |
                      (clk : clock(0)) | (msg : empty) | (hdl : h(0)) > .

  eq eagerEnabled(< f, r >) = eagerEnabled({f, r}) .
  rl [SiteCall] : { M(C) , (tr : t) | (msg : rho) | (hdl : h(N)) |
                          (clk : clock(T)) | r} =>
              [ ? h(N) , (tr : (t . M < C , h(N) | T >)) |
                      (msg : (rho [M, C, h(N)])) |
       (hdl : h(s N)) | (clk : clock(T)) | r] .
  eq eagerEnabled({ M(C) , (tr : t) | (msg : rho) | (hdl : h(N)) |
                                         (clk : clock(T)) | r}) = true .
  eq intActions({ M(C) , (tr : t) | (msg : rho) | (hdl : h(N)) |
                                         (clk : clock(T)) | r}) = true .


 ceq [M, C, h] = [self, app(M, C, rand), h] if M =/= self .


  rl [SiteRet] : { ? h , (tr : t) | (msg : (rho [self, c, h])) |
                         (clk : clock(T)) | r} =>
             [ ! c , (tr : (t . h ? c | T)) | (msg : rho) |
                                             (clk : clock(T)) | r] .
  eq eagerEnabled({ ? h , (tr : t) | (msg : (rho [self, c, h])) |
                                         (clk : clock(T)) | r}) = true .


  rl [Pub] : {! c , (tr : t) | (clk : clock(T)) | r}
              => [zero, (tr : t . (!! c | T)) | (clk : clock(T)) | r] .
  eq eagerEnabled({! c , (tr : t) | (clk : clock(T)) | r}) = true .
  eq intActions({! c , (tr : t) | (clk : clock(T)) | r}) = true .


  rl [Def] : { E(P), (tr : t) | (con : (sigma , E Q := f)) | r } =>
             [ ([Q <- P] f), (tr : t . tau) |
                                       (con : (sigma , E Q := f)) | r ] .
  eq eagerEnabled({ E(P), (tr : t) | (con : (sigma , E Q := f)) | r })
     = true .
  eq intActions({ E(P), (tr : t) | (con : (sigma , E Q := f)) | r })
     = true .


 crl [SymV]  : { f | g , (tr : t) | r}
                              => [ f' | g, (tr : t . (!! c | T)) | r']
if { f , (tr : nil) | r}
                                  => [ f' , (tr : (!! c | T)) | r' ] .

 crl [SymN1] : { f | g , (tr : t) | r} => [ f' | g, (tr : t . tau) | r']
if { f , (tr : nil) | r} => [ f' , (tr : tau) | r' ] .

 crl [SymN2] : { f | g , (tr : t) | r}
                               => [ f' | g, (tr : t . h ? c | T) | r']
             if intActions({ f | g , (tr : t) | r}) =/= true
```

```
           /\ { f , (tr : nil) | r} => [ f' , (tr : h ? c | T) | r' ] .

 crl [SymN3] : { f | g , (tr : t) | r}
                            => [ f' | g, (tr : t . M < C , h | T >) | r']
if { f , (tr : nil) | r}
                                => [ f' , (tr : M < C , h | T >) | r' ] .
 ceq eagerEnabled({ f | g , (tr : t) | r}) = true
                 if eagerEnabled({ f , (tr : nil) | r}) .
 ceq intActions({ f | g , (tr : t) | r}) = true
                  if  intActions({ f , (tr : nil) | r}) .


 crl [Seq1V] : { f > x > g , (tr : t) | r}
              => [ (f' > x > g) | ([x := c] g) , (tr : (t . tau)) | r' ]
if { f , (tr : nil) | r }
                  => [ f' , (tr : (nil . (!! c | T))) | r' ] .

 crl [Seq1N1] : { f > x > g , (tr : t) | r}
                                => [ f' > x > g , (tr : (t . tau)) | r' ]
if { f , (tr : nil) | r }
                                    => [ f' , (tr : (nil . tau)) | r' ] .

 crl [Seq1N2] : { f > x > g , (tr : t) | r}
                         => [ f' > x > g , (tr : (t . h ? c | T)) | r' ]
if intActions({ f > x > g , (tr : t) | r}) =/= true
/\ { f , (tr : nil) | r }
                   => [ f' , (tr : (nil . h ? c | T)) | r' ] .

 crl [Seq1N3] : { f > x > g , (tr : t) | r}
                   => [ f' > x > g , (tr : (t . M < C , h | T >)) | r' ]
if { f , (tr : nil) | r }
                       => [ f' , (tr : (nil . M < C , h | T >)) | r' ] .

 ceq eagerEnabled({ f > x > g , (tr : t) | r}) = true
                                if eagerEnabled({ f , (tr : nil) | r }) .
 ceq intActions({ f > x > g , (tr : t) | r}) = true
                                  if intActions({ f , (tr : nil) | r }) .

 crl [Asym1V] : { g where x :in f , (tr : t) | r}
                              => [ ([x := c] g) , (tr : (t . tau)) | r' ]
if { f , (tr : nil) | r }
                               => [ f' , (tr : (nil . !! c | T)) | r' ] .

 crl [Asym1N1] : { g where x :in f , (tr : t) | r}
                         => [ g where x :in f' , (tr : (t . tau)) | r' ]
if { f , (tr : nil) | r }
                                    => [ f' , (tr : (nil . tau)) | r' ] .

 crl [Asym1N2] : { g where x :in f , (tr : t) | r}
                   => [ g where x :in f' , (tr : (t . h ? c | T)) | r' ]
```

```
if intActions({ g where x :in f , (tr : t) | r}) =/= true
/\ { f , (tr : nil) | r }
                              => [ f' , (tr : (nil . h ? c | T)) | r' ] .

 crl [Asym1N3] : { g where x :in f , (tr : t) | r}
              => [ g where x :in f' , (tr : (t . M < C , h | T >)) | r' ]
                if { f , (tr : nil) | r }
                        => [ f' , (tr : (nil . M < C , h | T >)) | r' ] .

 ceq eagerEnabled({ g where x :in f , (tr : t) | r}) = true
                               if eagerEnabled({ f , (tr : nil) | r })  .
 ceq intActions({ g where x :in f , (tr : t) | r}) = true
                                 if intActions({ f , (tr : nil) | r })  .


 crl [Asym2V] : { g where x :in f , (tr : t) | r}
                     => [ g' where x :in f , (tr : (t . !! c | T)) | r' ]
if { g , (tr : nil) | r }
                                     => [ g' , (tr : (!! c | T)) | r' ] .
 crl [Asym2N1] : { g where x :in f , (tr : t) | r}
                          => [ g' where x :in f , (tr : (t . tau)) | r' ]
if { g , (tr : nil) | r } => [ g' , (tr : tau) | r' ] .
 crl [Asym2N2] : { g where x :in f , (tr : t) | r}
                    => [ g' where x :in f , (tr : (t . h ? c | T)) | r' ]
if intActions({ g where x :in f , (tr : t) | r}) =/= true
/\ { g , (tr : nil) | r }
                                     => [ g' , (tr : h ? c | T ) | r' ] .
 crl [Asym2N3] : { g where x :in f , (tr : t) | r}
              => [ g' where x :in f , (tr : (t . M < C , h | T >)) | r' ]
if { g , (tr : nil) | r }
                                => [ g' , (tr : M < C , h | T >) | r' ] .
 ceq eagerEnabled({ g where x :in f , (tr : t) | r}) = true
                                if eagerEnabled({ g , (tr : nil) | r }) .
 ceq intActions({ g where x :in f , (tr : t) | r}) = true
                                  if intActions({ g , (tr : nil) | r }) .

endm
```

A.1.6 Asynchronous Reduction Semantics

```
--- sync: traces, handle counter

mod ORC-SEMANTICS is
  inc RCONF-HC .
  pr SUBSTITUTION .
  pr SIMRANDOM .

  op initCon : DeclList -> Context .
```

```
 op [_] : Prog -> Conf .
 op app : SiteName ConstList Nat -> PreConst .
 op eagerEnabled : Conf -> [Bool] [frozen] .
 op eagerEnabled : Expr -> [Bool] [frozen] .
 op active : Expr -> [Bool] [frozen] .

 var d : Decl . var D : DeclList .
 var h h' : Handle .
 var M : SiteName . var c c' : Const .
 var x : Var . var ix : IVar .
 var P : AParamList . var C : ConstList .
 vars f f' g g' : Expr .
 var s : Event . var t : EventList .
 var n n' m : Nat .
 var E : ExprName . var Q : FParamList .
 var sigma : Context .
 var rho : MsgPool .
 vars r r' : Record .
 vars W W' : [Expr] . ---- #eq

 eq initCon(nilD) = mt .
 eq initCon(D ; d) = initCon(D) <- d .

 eq [D ; f] = < f , (tr : nil) | (con : initCon(D)) | (clk : clock(0)) |
                   (msg : empty) | (hdl : h(0)) > .


 --- error terms     ---- #eq
 op err : -> [Expr] .
 eq err | W = err .
 eq err > x > W = err .
 eq W > x > err = err .
 eq err where x :in W = err .
 eq W where x :in err = err .

 op tmph : -> Expr .
 eq S:Subst tmph = tmph .

 *** SiteCall
 ************
 op scallup : Expr SiteName ConstList -> [Expr] . ---- #eq
 op scalldn : Expr Handle -> Expr [frozen (1)] .

crl [SiteCall] : < f , r > => < scallup(f', M, C) , r >
                          if f => scallup(f', M, C) .    ---- #eq
 rl [SiteCall*] : M(C) => scallup(tmph, M, C) .

ceq scallup(f, M, C) | f' = scallup(f | f', M, C) if f' =/= zero .
 eq scallup(f, M, C) > x > f' = scallup(f > x > f', M, C) .
 eq scallup(f, M, C) where x :in f' = scallup(f where x :in f', M, C) .
 eq f' where x :in scallup(f, M, C) = scallup(f' where x :in f, M, C) .

 eq < scallup(f, M, C) , (tr : t) | (msg : rho) | (hdl : h(n)) | (clk : clock(m)) | r > =
      < scalldn(f, h(n)) , (tr : (t . M < C , h(n) | m >)) |
                         (msg : rho [M, C, h(n)]) | (hdl : h(s n)) |
(clk : clock(m)) | r > .

ceq scalldn(f | f', h) = scalldn(f, h) | scalldn(f', h)
      if f =/= zero /\ f' =/= zero .
 eq scalldn(f > x > f', h) = scalldn(f, h) > x > scalldn(f', h) .
 eq scalldn(f where x :in f', h) = scalldn(f, h) where x :in scalldn(f', h) .
 eq scalldn(zero, h) = zero .
 eq scalldn(M(P), h) = M(P) .
 eq scalldn(E(P), h) = E(P) .
 eq scalldn(! ix, h) = ! ix .
 eq scalldn(! c, h) = ! c .
 eq scalldn(? h', h) = ? h' .
 eq scalldn(tmph, h) = ? h .

 --- error terms
ceq scallup(W, M, C) = err               if W :: Expr == false .
ceq scallup(f, M, C) | W = err           if W :: Expr == false .
ceq scallup(f, M, C) > x > W = err       if W :: Expr == false .
ceq scallup(f, M, C) where x :in W = err if W :: Expr == false .
ceq W where x :in scallup(f, M, C) = err if W :: Expr == false .

ceq scalldn(W, h) = err               if W :: Expr == false .

 *** SiteRet
 ***********
 op sret : Expr Const Handle -> Expr [frozen(1)] .
 op _in_ : Handle Expr -> Bool [frozen (2)] .

crl [SiteRet] : < f , (tr : t) | (msg : (rho [self, c, h])) | (clk : clock(m)) | r >
               => < sret(f, c, h) , (tr : (t . h ? c | m)) |
                    (msg : rho) | (clk : clock(m)) | r >
  if h in f .

ceq sret(f | f', c, h) = sret(f, c, h) | sret(f', c, h)
       if f =/= zero /\ f' =/= zero .
 eq sret(f > x > f', c, h) = sret(f, c, h) > x > sret(f', c, h) .
 eq sret(f where x :in f', c, h) = sret(f, c, h) where x :in sret(f', c, h) .
 eq sret(zero, c, h) = zero .
 eq sret(M(P), c, h) = M(P) .
 eq sret(E(P), c, h) = E(P) .
 eq sret(! ix, c, h) = ! ix .
 eq sret(! c', c, h) = ! c' .
 eq sret(? h(n'), c, h(n)) = if (n' == n) then ! c else ? h(n') fi .

 ---- error terms
 ceq sret(W, c, h) = err if W :: Expr == false .

ceq h in (f | f') = h in f or h in f'
       if f =/= zero /\ f' =/= zero .
 eq h in (f > x > f') = h in f .
 eq h in (f where x :in f') = h in f or h in f' .
 eq h in zero = false .
 eq h in M(P) = false .
 eq h in E(P) = false .
 eq h in ! ix = false .
 eq h in ! c = false .
 eq h(n) in ? h(n') = if (n' == n) then true else false fi .

 *** Pub
 *******
 op pub : Expr Const -> [Expr]  .
 op pubTau : Expr -> [Expr] .

crl [Pub]    : < f, r > => < pub(f', c) , r > if f => pub(f', c) .
crl [PubTau] : < f, r > => < pubTau(f') , r > if f => pubTau(f') .
 rl [Pub*] : ! c => pub(zero, c) .

ceq pub(f, c) | f' = pub(f | f', c) if f' =/= zero .
 eq pub(f, c) > x > f' = pubTau(f > x > f' | ([x := c] f')) .
 eq pub(f, c) where x :in f' = pub(f where x :in f', c) .
 eq f' where x :in pub(f, c) = pubTau([x := c] f') .

ceq pubTau(f) | f' = pubTau(f | f') if f' =/= zero .
 eq pubTau(f) > x > f' = pubTau(f > x > f') .
 eq pubTau(f) where x :in f' = pubTau(f where x :in f') .
 eq f' where x :in pubTau(f) = pubTau(f' where x :in f) .

 eq < pub(f, c) , (tr : t) | (clk : clock(m)) | r > =
      < f , (tr : t . (!! c | m)) | (clk : clock(m)) | r > .
 eq < pubTau(f) , (tr : t) | r > = < f , (tr : t . tau) | r > .

 ---- error terms
ceq pub(W, c) = err               if W :: Expr == false .
ceq pub(f, c) | W = err           if W :: Expr == false .
ceq pub(f, c) > x > W = err       if W :: Expr == false .
ceq pub(f, c) where x :in W = err if W :: Expr == false .
ceq W where x :in pub(f, c) = err if W :: Expr == false .

ceq pubTau(W) = err               if W :: Expr == false .
ceq pubTau(f) | W = err           if W :: Expr == false .
ceq pubTau(f) > x > W = err       if W :: Expr == false .
ceq pubTau(f) where x :in W = err if W :: Expr == false .
ceq W where x :in pubTau(f) = err if W :: Expr == false .
```

```
 *** Expr Call
 *************
 op ecallup : Expr ExprName AParamList -> [Expr] .
 op ecalldn : Expr Expr -> Expr [frozen] .

crl [ExprCall] : < f , r > => < ecallup(f', E, P) , r > if f => ecallup(f', E, P) .
 rl [ExprCall*] : E(P) => ecallup(tmph, E, P) .

ceq ecallup(f, E, P) | f' = ecallup(f | f', E, P) if f' =/= zero .
 eq ecallup(f, E, P) > x > f' = ecallup(f > x > f', E, P) .
 eq ecallup(f, E, P) where x :in f' = ecallup(f where x :in f', E, P) .
 eq f' where x :in ecallup(f, E, P) = ecallup(f' where x :in f, E, P) .

 eq < ecallup(f, E, P) , (tr : t) | (con : (sigma , E Q := g)) | r > =
      < ecalldn(f, ([Q <- P] g)) , (tr : t . tau) | (con : (sigma , E Q := g)) | r > .

ceq ecalldn(f | f', g) = ecalldn(f, g) | ecalldn(f', g)
      if f =/= zero /\ f' =/= zero .
 eq ecalldn(f > x > f', g) = ecalldn(f, g) > x > ecalldn(f', g) .
 eq ecalldn(f where x :in f', g) = ecalldn(f, g) where x :in ecalldn(f', g) .
 eq ecalldn(zero, g) = zero .
 eq ecalldn(M(P), g) = M(P) .
 eq ecalldn(E(P), g) = E(P) .
 eq ecalldn(! ix, g) = ! ix .
 eq ecalldn(! c, g) = ! c .
 eq ecalldn(? h, g) = ? h .
 eq ecalldn(tmph, g) = g .

 ---- error terms
ceq ecallup(W, E, P) = err               if W :: Expr == false .
ceq ecallup(f, E, P) | W = err           if W :: Expr == false .
ceq ecallup(f, E, P) > x > W = err       if W :: Expr == false .
ceq ecallup(f, E, P) where x :in W = err if W :: Expr == false .
ceq W where x :in ecallup(f, E, P) = err if W :: Expr == false .

ceq ecalldn(W, g) = err               if W :: Expr == false .


 ---- simulating responses
ceq [M, C, h] = [self, app(M, C, rand), h] if M =/= self .


 *** Active Expressions
ceq active(f | f') = active(f) == true  or active(f') == true
       if f =/= zero /\ f' =/= zero .
 eq active(f > x > f') = active(f) .
 eq active(f where x :in f') = active(f) == true or  active(f') == true .
 eq active(M(C)) = true .
 eq active(! c) = true .
 eq active(E(P)) = true .
```

```
  *** Eager Configurations
 ceq eagerEnabled(< f , (msg : (rho [self, c, h])) | r >) = true if h in f .
 ceq eagerEnabled(< f , r >) = true if active(f) .

endm
```

A.1.7 Synchronous Reduction Semantics

Other than the change noted in section 6.3 for the site return rule, the synchronous reduction semantics specification is the same as the one for the asynchronous semantics given in Appendix A.1.6 above.

A.1.8 Sites and Timing

```
--- STANDARD SITES
mod SIGNAL-SITE is
  inc ORC-SEMANTICS .
  op signal : -> SiteName [ctor] .
  eq app(signal, nilA, n:Nat) = sig .
endm

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

mod LET-SITE is
  inc ORC-SEMANTICS .
  op let : -> SiteName [ctor] .
  eq app(let, c:Const, n:Nat) = c:Const .
  eq app(let, C:ConstList, n:Nat) = tup(C:ConstList) [owise] .
endm

mod CLOCK-SITE is
  inc ORC-SEMANTICS .
  op clock : -> SiteName [ctor] .
  var f : Expr . var r : Record .
  var rho : MsgPool . vars n m : Nat .
  var h : Handle .
  eq < f , r | (msg : (rho [self, app(clock, nilA, n), h])) | (clk : clock(m)) > =
     < f , r | (msg : (rho [self, m, h])) | (clk : clock(m)) > .
endm

mod ATIMER-SITE is
  inc ORC-SEMANTICS .
  op atimer : -> SiteName [ctor] .
  var f : Expr . var r : Record .
  var rho : MsgPool . vars n m m' : Nat .
  var h : Handle .
 ceq < f , r | (msg : (rho [self, app(atimer, m, n), h])) | (clk : clock(m')) > =
     < f , r | (msg : (rho [self, sig, h])) | (clk : clock(m')) > if m == m' .
endm

mod RTIMER-SITE is
  inc ORC-SEMANTICS .
  op rtimer : -> SiteName [ctor] .
  eq app(rtimer, 0, n:Nat) = sig .
endm

*** NON-STANDARD SITES
mod ARITHMETIC-SITES is
  inc ORC-SEMANTICS .
  ops add sub mul div : -> SiteName [ctor] .
  vars m n  : Nat .
  eq app(add, (n, m), 0) = n + m .
  eq app(sub, (n, m), 0) = if m < n then n - m else 0 fi .
  eq app(mul, (n, m), 0) = n * m .
  eq app(div, (n, m), 0) = if m > 0 then n quo m else 0 fi .
endm

mod BINRELATION-SITES is
  inc ORC-SEMANTICS .
  ops lt gt le ge eq ne : -> SiteName [ctor] .
  vars m n : Nat .
  eq app(lt, (n, m), 0) = tr(n < m) .
  eq app(gt, (n, m), 0) = tr(n > m) .
  eq app(le, (n, m), 0) = tr(n <= m) .
  eq app(ge, (n, m), 0) = tr(n >= m) .
  eq app(eq, (n, m), 0) = tr(n == m) .
  eq app(ne, (n, m), 0) = tr(n =/= m) .
endm

mod LOGICAL-SITES is
  inc ORC-SEMANTICS .
  ops not and or : -> SiteName [ctor] .
  vars B B' : Bool .
  eq app(not, tr(B), 0) = tr(not B) .
  eq app(and, (tr(B), tr(B')), 0) = tr(B and B') .
  eq app(or, (tr(B), tr(B')), 0) = tr(B or B') .
endm

mod TUPLE-SITES is
  inc ORC-SEMANTICS .
  ops head tail empty : -> SiteName [ctor] .

  var c : Const . var C : ConstList .
  eq app(head, tup(c,C), 0) = c .
  eq app(tail, tup(c,C), 0) = C .

  eq app(empty, tup(nilA), 0) = tr(true) .
  eq app(empty, tup(c,C), 0) = tr(false) .
endm

mod RT-SEMANTICS is
  inc SIGNAL-SITE .
  inc IF-SITE .
  inc LET-SITE .
  inc CLOCK-SITE .
  inc ATIMER-SITE .
  inc RTIMER-SITE .
  inc ARITHMETIC-SITES .
  inc BINRELATION-SITES .
  inc LOGICAL-SITES .
  inc TUPLE-SITES .

  var h : Handle . var n m : Nat . var fld : Field . var Msg : Msg .
  var M : SiteName . var c : Const . var C : ConstList .
  var rho : MsgPool . var r : Record .  var f : Expr .

 crl [Tick] : < f, (clk : clock(m)) | r > => < f, (clk : clock(s m)) | delta(r) >
                  if eagerEnabled(< f, (clk : clock(m)) | r >) =/= true .

  op delta : Record -> Record .
  op delta : MsgPool -> MsgPool .

  eq delta(null) = null .
 ceq delta(fld | r) = delta(fld) | delta(r) if r =/= null .
  eq delta(msg : rho) = msg : delta(rho) .
  eq delta(fld) = fld [owise] .

  eq delta(empty) = empty .
 ceq delta(Msg rho) = delta(Msg) delta(rho) if rho =/= empty .
  eq delta([self,app(rtimer, s(n), m), h]) = [self, app(rtimer, n, m), h] . --- ignore delay
  eq delta([self,app(M, C, s(n)), h]) = [self, app(M, C, n), h] .
  eq delta([self,app(M, C, 0), h]) = [self, app(M, C, 0), h] .
  --- needed for the case when a site doesn't respond
  eq delta([M,C,h]) = [M,C,h] [owise] .
endm
```

A.1.9 Dining Philosophers Specification

```
mod SiteNameSet is
  inc RT-SEMANTICS .
  sort SiteNameSet .
  subsort SiteName < SiteNameSet .
  op mts : -> SiteNameSet [ctor] .
  op _._ : SiteNameSet SiteNameSet -> SiteNameSet [ctor assoc comm
                                                              id: mts] .

  op _in_ : SiteName SiteNameSet -> Bool .

  var M : SiteName .
  var MX : SiteNameSet .

  eq M in M . MX = true .
  eq M in MX = false [owise] .
endm

mod DINING-PHILOSOPHERS is
  inc SiteNameSet .

  --- sites syntax
  op eat : -> SiteName [ctor] .

  op fork[_] : Nat -> SiteName [ctor] .

  --- new constants
  ops put get : -> Const [ctor] .

  --- extension of configurations
  op t-forks : -> Index [ctor] .
  op _:_ : Index SiteNameSet -> Field [ctor] .

  --- dining philosophers sites behavior
  var f : Expr . var r : Record .
  var rho : MsgPool . var h : Handle .
  vars i m n : Nat .
  var MX : SiteNameSet .

  --- fork get request
 ceq < f , r | (msg : (rho [self, app(fork[i], get, 0), h])) |
                                                     (t-forks : MX) > =
     < f , r | (msg : (rho [self, sig, h])) | (t-forks : MX . fork[i]) >
       if not (fork[i] in MX) .

  --- fork put request
  eq < f , r | (msg : (rho [self, app(fork[i], put, 0), h])) |
                                          (t-forks : MX . fork[i]) > =
     < f , r | (msg : (rho [self, sig, h])) | (t-forks : MX) > .

  --- eat request
  eq app(eat, nilA, 0) = sig .

endm

mod DF-PROGRAM is
```

```
 inc DINING-PHILOSOPHERS .

 --- philosopher expression names
 op phil[_] : Nat -> ExprName [ctor] .

 --- construct an Orc program with the given number of philosophers
 op df : NzNat -> Prog .
 op df-decl : Nat NzNat -> DeclList .

 op df-df : NzNat -> Prog .
 op df-decl-df : Nat NzNat -> DeclList .

 op df-exp : Nat -> Expr .

 var n : NzNat .
 var m : Nat .

ceq df(n) = df-decl(n - 1,n) ; df-exp(n - 1) if n > 1 .

 eq df-exp(s(m)) = phil[s(m)]() | df-exp(m) .
 eq df-exp(0) = phil[0]() .

 eq df-decl(s(m), n) = phil[s(m)] nilF :=
       fork[s(m)](get) > 'vt > fork[(s(m) + 1) rem n](get) > 'vt >
       eat() > 'vt >
       fork[s(m)](put) > 'vt > fork[(s(m) + 1) rem n](put) > 'vt >
       phil[s(m)]() ; df-decl(m, n) .
 eq df-decl(0, n) =  phil[0] nilF :=
       fork[0](get) > 'vt > fork[1](get) > 'vt >
       eat() > 'vt >
       fork[0](put) > 'vt > fork[1](put) > 'vt >
       phil[0]() .

ceq df-df(n) = df-decl-df(n - 1,n) ; df-exp(n - 1) if n > 1 .

 eq df-decl-df(s(m), n) = phil[s(m)] nilF :=
       fork[s(m)](get) > 'vt > fork[(s(m) + 1) rem n](get) > 'vt >
       eat() > 'vt >
       fork[s(m)](put) > 'vt > fork[(s(m) + 1) rem n](put) > 'vt >
       phil[s(m)]() ; df-decl-df(m, n) .

 eq df-decl-df(0, n) =  phil[0] nilF :=
       fork[1](get) > 'vt > fork[0](get) > 'vt >
       eat() > 'vt >
       fork[0](put) > 'vt > fork[1](put) > 'vt >
       phil[0]() .

 var f : Expr . var D : DeclList .
 --- constructs an initial configuration for the df problem with the
 --- trace field
```

```
  op {_} : Prog -> Conf .
  eq {D ; f} = < f , (tr : nil) | (con : initCon(D)) | (clk : clock(0)) |
                     (msg : empty) | (hdl : mth) | (t-forks : mts) > .

  --- constructs an initial configuration for the df problem *without*
  --- the trace field
  op {_}* : Prog -> Conf .
  eq {D ; f}* = < f , (con : initCon(D)) | (clk : clock(0)) |
                     (msg : empty) | (hdl : mth) | (t-forks : mts) > .
endm

mod DF-PREDS is
  pr DF-PROGRAM .
  inc SATISFACTION .

  subsort Conf < State .

  op enabled : -> Prop .
  op eats : Nat -> Prop .

  var C : Conf .
  var i : Nat .
  vars x x' : Var .
  vars f f' : Expr .
  var r : Record .

 ceq < f , r > |= enabled = true if eagerEnabled(< f , r >) .
  eq < (eat() > x > f) > x' > phil[i]() | f', r > |= eats(i) = true .
endm

mod DF-CHECK is
  pr DF-PREDS .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
  op init : -> Conf .
  op rel-excl : NzNat -> Formula .
  op re : Nat NzNat -> Formula .
  op no-deadlock : -> Formula .

  var n : NzNat .
  var m : Nat .
  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)) .

  eq no-deadlock = [] enabled .
endm
```

A.2 The Distributed Object-based Semantics

A.2.1 Syntax and CINNI Substitution Calculus

See appendices A.1.1 and A.1.2.

A.2.2 Objects and Messages

```
fmod CLOCK is
  pr NAT .
  sort ClockAttr .
  op halt : -> ClockAttr [ctor] .
  op c : Nat -> ClockAttr [ctor] .
  eq c(20) = halt . --- finite state clock
endfm

mod ORC-CLOCK is
  pr CLOCK .
  inc CONFIGURATION .

  op Clock : -> Cid [ctor] .
  op clk:_ : ClockAttr -> Attribute [ctor gather (&)] .
  ops C1 : -> Oid [ctor] .

endm


fmod HANDLE-SET is
  pr ORC-EXTENDED-SYNTAX .
  sort HandleSet .
  subsort Handle < HandleSet .

  op mth : -> HandleSet [ctor] .
  op _#_ : HandleSet HandleSet -> HandleSet [ctor assoc comm id: mth] .

  vars h h' : Handle .
  vars H : HandleSet .
  var n : Nat .
  vars f f' : Expr .
  var M : SiteName . var E : ExprName . var u : Builtin .
  var P : AParamList .
  var ix : IVar . var x : Var . var c : Const .

  eq h # h # H = h # H .

  op _usedin_ : Handle HandleSet -> Bool .
  eq h usedin (h' # H) = h == h' or h usedin H .
  eq h usedin mth = false .

  op gFresh : HandleSet -> Handle .
  op gFreshAux : HandleSet Handle -> Handle .

  eq gFresh(H) = gFreshAux(H, h(0)) .
  eq gFreshAux(H, h(n)) = if h(n) usedin H
                          then gFreshAux(H, h(s n))
                          else h(n) fi .

  op handles : Expr -> HandleSet [frozen (1)] .
 ceq handles(f | f') = handles(f) # handles(f')
       if f =/= zero /\ f' =/= zero .
  eq handles(f > x > f') = handles(f) # handles(f') .
  eq handles(f where x :in f') = handles(f) # handles(f') .
  eq handles(zero)  = mth .
  eq handles(M(P))  = mth .
  eq handles(E(P))  = mth .
  eq handles(u(P))  = mth .
  eq handles(ix(P)) = mth .
  eq handles(! ix)  = mth .
  eq handles(! c)   = mth .
  eq handles(? h)     = h .

endfm

fmod CONTEXT is
  protecting ORC-EXTENDED-SYNTAX .
  sort Context .
  subsort Decl < Context .

  op mt : -> Context [ctor] .
  op _,_ : Context Context -> Context [ctor assoc comm id: mt prec 42] .
  op _<-_ : Context Decl -> Context [ctor prec 45 gather (E e)] .

  var E : ExprName .
  var f f' : Expr .
  var sigma : Context .
  var d : Decl .
  var Q Q' : FParamList .

  eq E Q := f , sigma <- E Q' := f' = E Q' := f' , sigma .
  eq sigma <- d = d , sigma [owise] .
endfm

mod ORC-EXPR is
  pr ORC-EXTENDED-SYNTAX .
  pr HANDLE-SET .
  pr CONTEXT .
  inc CONFIGURATION .

  op Expr : -> Cid [ctor] .
  op E : Nat -> Oid [ctor] .
  op exp:_ : Expr -> Attribute [ctor gather (&) frozen] .  ----#eq
  op con:_ : Context -> Attribute [ctor gather (&)] .
  op hdl:_ : HandleSet -> Attribute [ctor gather (&)] .



endm


mod ORC-SITE is
  pr ORC-EXTENDED-SYNTAX .
  inc CONFIGURATION .

  sorts Op OState .
  op free : -> Op [ctor] .
  op exec : ConstList Handle Oid -> Op .

  op nil : -> OState [ctor] .
  op __ : OState OState -> OState [ctor assoc id: nil] .

  op Site : -> Cid [ctor] .
  op name:_ : SiteName -> Attribute [ctor gather (&)] .
  op op:_ : Op -> Attribute [ctor gather (&)] .
  op state:_ : OState -> Attribute [ctor gather (&)] .

  --- standard sites
  ops S : Nat -> Oid [ctor] .
  ops let if clock rtimer atimer signal : -> SiteName [ctor] .

endm

mod ORC-MESSAGE is
  pr ORC-EXTENDED-SYNTAX .
  inc CONFIGURATION .

  sort Content .

  op _<-_ : Oid Content -> Msg [ctor prec 15] .
  op _<-_ : SiteName Content -> Msg [ctor prec 15] .

  op sr : Const Handle Nat -> Content [ctor] .
  op sc : Oid ConstList Handle Nat -> Content [ctor] .

endm
```

A.2.3 Distributed Semantics

```
mod SIMCOUNTER is
  pr NAT .
  op counter : -> [Nat] .
  rl [count] : counter => s(counter) .
  rl [eval]  : counter => 0 .
  eq s_^5(counter) = counter .  --- finite state counter
endm

mod ORC-RWSEM is
  pr ORC-EXPR .
  pr ORC-SITE .
  pr ORC-CLOCK .
  pr ORC-MESSAGE .
  pr SUBSTITUTION .
  pr RANDOM .
  pr SIMCOUNTER .
  pr RAT .
  pr CONVERSION .

  sort System .

  var OE OS OT OC : Oid .
  vars AS AS' : AttributeSet .
  vars CF CF' : Configuration .
  var d : Decl . var D : DeclList .
  var h h' : Handle . var H H' : HandleSet .
  var M : SiteName . var c c' : Const .
  var x : Var . var ix : IVar .
  var P : AParamList . var C : ConstList .
  vars f f' g g' : Expr .
  var n n' m : Nat .
  var E : ExprName . var Q : FParamList .
  var sigma : Context .
  var u : Builtin .
  vars W W' : [Expr] . ---- #eq

  op {_} : Configuration -> System [ctor] .
  op active : Expr -> [Bool] [frozen] .
  op eagerEnabled : System -> [Bool] [frozen] .

  op rand : -> [Nat] .
  ---rl [rnd] : rand => floor((random(counter) / 4294967296) * 10) .
  eq rand = 0 .

  --- error terms
  op err : -> [Expr] .
  eq err | W = err .
  eq err > x > W = err .
  eq W > x > err = err .
  eq err where x :in W = err .
  eq W where x :in err = err .
```

```
 *** SiteCall
 ************
 op scallup : Expr SiteName ConstList -> [Expr] .
 op scalldn : Expr Handle -> Expr [frozen (1)] .
 op tmph : -> Expr .

 eq S:Subst tmph = tmph .

crl [SiteCall] : < OE : Expr | exp: f , AS >
                 => < OE : Expr | exp: scallup(f', M, C) , AS >
                       if f => scallup(f', M, C) .    ---- #eq
 rl [SiteCall*] : M(C) => scallup(tmph, M, C) .

ceq scallup(f, M, C) | f' = scallup(f | f', M, C) if f' =/= zero .
 eq scallup(f, M, C) > x > f' = scallup(f > x > f', M, C) .
 eq scallup(f, M, C) where x :in f' = scallup(f where x :in f', M, C) .
 eq f' where x :in scallup(f, M, C) = scallup(f' where x :in f, M, C) .

ceq < OE : Expr | exp: scallup(f, M, C) , hdl: H , AS >
      = < OE : Expr | exp: scalldn(f, h) , hdl: h # H , AS >
        M <- sc(OE, C, h, 0)
           if h := gFresh(H) .

ceq scalldn(f | f', h) = scalldn(f, h) | scalldn(f', h)
      if f =/= zero /\ f' =/= zero .
 eq scalldn(f > x > f', h) = scalldn(f, h) > x > scalldn(f', h) .
 eq scalldn(f where x :in f', h) = scalldn(f, h) where x :in scalldn(f', h) .
 eq scalldn(zero, h) = zero .
 eq scalldn(M(P), h) = M(P) .
 eq scalldn(E(P), h) = E(P) .
 eq scalldn(u(P), h) = u(P) .
 eq scalldn(ix(P), h) = ix(P) .   ---- newly added
 eq scalldn(! ix, h) = ! ix .
 eq scalldn(! c, h) = ! c .
 eq scalldn(? h', h) = ? h' .
 eq scalldn(tmph, h) = ? h .

 --- error terms
ceq scallup(W, M, C) = err               if W :: Expr == false .
ceq scallup(f, M, C) | W = err           if W :: Expr == false .
ceq scallup(f, M, C) > x > W = err       if W :: Expr == false .
ceq scallup(f, M, C) where x :in W = err if W :: Expr == false .
ceq W where x :in scallup(f, M, C) = err if W :: Expr == false .

ceq scalldn(W, h) = err               if W :: Expr == false .

 *** SiteRet
 ***********
 op sret : Expr Const Handle -> Expr [frozen(1)] .
 op _in_ : Handle Expr -> Bool [frozen (2)] .
```

```
crl [SiteRet] : OE <- sr(c, h, 0)
                < OE : Expr | exp: f , hdl: h # H , AS >
                => < OE : Expr | exp: sret(f, c, h) , hdl: H , AS >
   if h in f .

ceq sret(f | f', c, h) = sret(f, c, h) | sret(f', c, h)
       if f =/= zero /\ f' =/= zero .
 eq sret(f > x > f', c, h) = sret(f, c, h) > x > sret(f', c, h) .
 eq sret(f where x :in f', c, h) = sret(f, c, h) where x :in sret(f', c, h) .
 eq sret(zero, c, h) = zero .
 eq sret(M(P), c, h) = M(P) .
 eq sret(E(P), c, h) = E(P) .
 eq sret(ix(P), c, h) = ix(P) .   ---- newly added
 eq sret(u(P), c, h) = u(P) .
 eq sret(! ix, c, h) = ! ix .
 eq sret(! c', c, h) = ! c' .
 eq sret(? h(n'), c, h(n)) = if (n' == n) then ! c else ? h(n') fi .

 ---- error terms
 ceq sret(W, c, h) = err if W :: Expr == false .

ceq h in (f | f') = h in f or h in f'
       if f =/= zero /\ f' =/= zero .
 eq h in (f > x > f') = h in f .
 eq h in (f where x :in f') = h in f or h in f' .
 eq h in zero = false .
 eq h in M(P) = false .
 eq h in E(P) = false .
 eq h in u(P) = false .
 eq h in ix(P) = false .    ---- newly added
 eq h in ! ix = false .
 eq h in ! c = false .
 eq h(n) in ? h(n') = if (n' == n) then true else false fi .

 *** Pub
 *******
 op pub : Expr Const -> [Expr] .
 op pubTau : Expr -> [Expr] .

crl [Pub]    : < OE : Expr | exp: f, AS >
               => < OE : Expr | exp: pub(f', c) , AS > if f => pub(f', c) .
crl [PubTau] : < OE : Expr | exp: f, AS >
               => < OE : Expr | exp: pubTau(f') , AS > if f => pubTau(f') .
 rl [Pub*] : ! c => pub(zero, c) .

ceq pub(f, c) | f' = pub(f | f', c) if f' =/= zero .
 eq pub(f, c) > x > f' = f > x > f' | ([x := c] f') .
 eq pub(f, c) where x :in f' = pub(f where x :in f', c) .
 eq f' where x :in pub(f, c) = [x := c] f' .
 eq < OE : Expr | exp: pub(f, c) , AS >
    = < OE : Expr | exp: f , AS > .

ceq pubTau(f) | f' = pubTau(f | f') if f' =/= zero .
 eq pubTau(f) > x > f' = pubTau(f > x > f') .
 eq pubTau(f) where x :in f' = pubTau(f where x :in f') .
 eq f' where x :in pubTau(f) = pubTau(f' where x :in f) .

 eq < OE : Expr | exp: pubTau(f) , AS > = < OE : Expr | exp: f , AS > .

 ---- error terms
ceq pub(W, c) = err               if W :: Expr == false .
ceq pub(f, c) | W = err           if W :: Expr == false .
ceq pub(f, c) > x > W = err       if W :: Expr == false .
ceq pub(f, c) where x :in W = err if W :: Expr == false .
ceq W where x :in pub(f, c) = err if W :: Expr == false .

ceq pubTau(W) = err               if W :: Expr == false .
ceq pubTau(f) | W = err           if W :: Expr == false .
ceq pubTau(f) > x > W = err       if W :: Expr == false .
ceq pubTau(f) where x :in W = err if W :: Expr == false .
ceq W where x :in pubTau(f) = err if W :: Expr == false .

 *** Expr Call
 *************
 op ecallup : Expr ExprName AParamList -> [Expr] .
 op ecalldn : Expr Expr -> Expr [frozen] .

crl [ExprCall] : < OE : Expr | exp: f , AS >
                 => < OE : Expr | exp: ecallup(f', E, P) , AS >
                   if f => ecallup(f', E, P) .
 rl [ExprCall*] : E(P) => ecallup(tmph, E, P) .

ceq ecallup(f, E, P) | f' = ecallup(f | f', E, P) if f' =/= zero .
 eq ecallup(f, E, P) > x > f' = ecallup(f > x > f', E, P) .
 eq ecallup(f, E, P) where x :in f' = ecallup(f where x :in f', E, P) .
 eq f' where x :in ecallup(f, E, P) = ecallup(f' where x :in f, E, P) .

 eq < OE : Expr | exp: ecallup(f, E, P) , con: (sigma , E Q := g) , AS > =
      < OE : Expr | exp: ecalldn(f, ([Q <- P] g)) , con: (sigma , E Q := g) , AS > .

ceq ecalldn(f | f', g) = ecalldn(f, g) | ecalldn(f', g)
      if f =/= zero /\ f' =/= zero .
 eq ecalldn(f > x > f', g) = ecalldn(f, g) > x > ecalldn(f', g) .
 eq ecalldn(f where x :in f', g) = ecalldn(f, g) where x :in ecalldn(f', g) .
 eq ecalldn(zero, g) = zero .
 eq ecalldn(M(P), g) = M(P) .
 eq ecalldn(E(P), g) = E(P) .
 eq ecalldn(u(P), g) = u(P) .
```

```
  eq ecalldn(ix(P), g) = ix(P) .   ---- newly added
  eq ecalldn(! ix, g) = ! ix .
  eq ecalldn(! c, g) = ! c .
  eq ecalldn(? h, g) = ? h .
  eq ecalldn(tmph, g) = g .

  ---- error terms
 ceq ecallup(W, E, P) = err               if W :: Expr == false .
 ceq ecallup(f, E, P) | W = err           if W :: Expr == false .
 ceq ecallup(f, E, P) > x > W = err       if W :: Expr == false .
 ceq ecallup(f, E, P) where x :in W = err if W :: Expr == false .
 ceq W where x :in ecallup(f, E, P) = err if W :: Expr == false .

 ceq ecalldn(W, g) = err               if W :: Expr == false .

  *** Site consuming a call

  eq  M <- sc(OE, C, h, 0) < OS : Site | name: M , op: free, AS >
    = < OS : Site | name: M , op: exec(C, h, OE) , AS > .

  *** Remove suprious responses
 ----ceq < OE : Expr | hdl: H , AS > OE <- sr(c, h, n)
 ----    = < OE : Expr | hdl: H , AS > if not(h usedin H) .


  *** Active Expressions
 ceq active(f | f') = active(f) == true  or active(f') == true
       if f =/= zero /\ f' =/= zero .
  eq active(f > x > f') = active(f) .
  eq active(f where x :in f') = active(f) == true or  active(f') == true .
  eq active(M(C)) = true .
  eq active(! c) = true .
  eq active(E(P)) = true .

  *** Eager Systems
 ceq eagerEnabled({OE <- sr(c, h, 0)
                  < OE : Expr | exp: f , AS > CF}) = true
     if h in f .
  eq eagerEnabled({M <- sc(OE, C, h, 0) < OS : Site | name: M , op: free, AS > CF}) = true .
 ceq eagerEnabled({< OE : Expr | exp: f , AS > CF}) = true if active(f) .

endm

mod ORC-SITES is
  inc ORC-RWSEM .

  var OE OS OT OC : Oid .
  vars AS AS' : AttributeSet .
  vars CF CF' : Configuration .
  var d : Decl . var D : DeclList .
  var h h' : Handle . var H : HandleSet .
  var M : SiteName . var c c' : Const .
  var x : Var . var ix : IVar .
  var P : AParamList . var C : ConstList .
  vars f f' g g' : Expr .
  var n n' m t t' : Nat .
  var E : ExprName . var Q : FParamList .
  var sigma : Context .

  --- ops to setup initial configuration
  op initCon : DeclList -> Context .
  op [_:_] : Oid Prog -> Configuration .
  op stdSites : -> Configuration .
  op initClk : -> Configuration .
  op $_$ : Configuration -> System .

  eq initCon(nilD) = mt .
  eq initCon(D ; d) = initCon(D) <- d .

  eq [OE : D ; f] = < OE : Expr |
                    exp: f , con: initCon(D) , hdl: mth > .

  eq stdSites = < S(1) : Site | name: let ,op: free , state: nil >
               < S(2) : Site | name: if ,  op: free , state: nil >
< S(3) : Site | name: clock ,  op: free , state: nil >
< S(4) : Site | name: rtimer , op: free , state: nil >
< S(5) : Site | name: atimer , op: free , state: nil >
< S(6) : Site | name: signal , op: free , state: nil > .

  eq initClk = < C1 : Clock | clk: c(0) > .

  eq $ CF $ = {CF stdSites initClk} .

  --- standard sites behavior
  --- let
  eq < OS : Site | name: let, op: exec(c, h, OE) , AS > =
       < OS : Site | name: let , op: free , AS >
       OE <- sr(c, h, rand) .
  eq < OS : Site | name: let, op: exec(C, h, OE) , AS > =
       < OS : Site | name: let , op: free , AS >
       OE <- sr(tup(C), h, rand) [owise] .

  --- if
  eq < OS : Site | name: if, op: exec(tr(true), h, OE) , AS > =
       < OS : Site | name: if , op: free , AS >
       OE <- sr(sig, h, rand) .

  eq < OS : Site | name: if, op: exec(tr(false), h, OE) , AS > =
       < OS : Site | name: if , op: free , AS > .
  --- clock
  eq < OS : Site | name: clock, op: exec(nilA, h, OE) , AS > < OC : Clock | clk: c(t) > =
       < OS : Site | name: clock , op: free , AS > < OC : Clock | clk: c(t) >
       OE <- sr(t, h, 0) .

  --- rtimer
  eq < OS : Site | name: rtimer, op: exec(n, h, OE) , AS > =
       < OS : Site | name: rtimer , op: free , AS >
       OE <- sr(sig, h, n) .

  --- atimer
  eq < OS : Site | name: atimer, op: exec(t, h, OE) , AS > < OC : Clock | clk: c(t') > =
       < OS : Site | name: atimer , op: free , AS > < OC : Clock | clk: c(t') >
       OE <- sr(sig, h, t - t') .

  --- signal
  eq < OS : Site | name: signal, op: exec(nilA, h, OE) , AS > =
       < OS : Site | name: signal , op: free , AS >
       OE <- sr(sig, h, 0) .

  ---- silent sites
  eq if(tr(false)) = zero .
endm

mod ORC-RWTIME is
  inc ORC-SITES .

  var OE OS OT OC : Oid .
  vars AS AS' : AttributeSet .
  vars CF CF' : Configuration .
  var h : Handle .
  var M : SiteName . var c c' : Const .
  var C : ConstList .   var t d : Nat .
  var o : Object . var m : Msg .

  op delta : Configuration -> Configuration [frozen] .

  crl [tick] :
  {< OC : Clock | clk: c(t) , AS > CF} =>
    {< OC : Clock | clk: c(s(t)) , AS > delta(CF)}
       if eagerEnabled({< OC : Clock | clk: c(t) , AS > CF}) =/= true .

  eq delta(none) = none .
  ceq delta(o CF) = delta(o) delta(CF) if CF =/= none .
  ceq delta(m CF) = delta(m) delta(CF) if CF =/= none .

  eq delta(< OS : Site | name: rtimer , op: exec(s(d), h, OE), AS >) =
       < OS : Site | name: rtimer , op: exec(d, h, OE), AS > .

  eq delta(M <- sc(OE, C, h, s(d))) = M <- sc(OE, C, h, d) .
  eq delta(OE <- sr(c, h, s(d))) = OE <- sr(c, h, d) .

  eq delta(o) = o [owise] .
  eq delta(m) = m [owise] .
endm
```

A.2.4 Non-standard Sites and Examples

```
mod ORC-INTERFACE is
  inc ORC-RWTIME .

  op get : -> Const [ctor] .
  op put : -> Const [ctor] .
endm

mod ORC-COUNTER-SITE is
  inc ORC-INTERFACE .

  ops inc reset : -> Const [ctor] .

  vars OS OE OE' : Oid .
  vars n i : Nat .
  vars h : Handle .
  var AS : AttributeSet .

  ****************
  *** Counter Site
  op SCNT : -> Oid [ctor] .
  op count : -> SiteName [ctor] .
  op count : Nat -> OState [ctor] .
  *** increment message
  eq < OS : Site | name: count, op: exec(inc, h, OE) , state: count(n) , AS > =
       < OS : Site | name: count , op: free , state: count(s(n)) , AS >
       OE <- sr(sig, h, 0) .
  *** get message
  eq < OS : Site | name: count, op: exec(get, h, OE) , state: count(n) , AS > =
       < OS : Site | name: count , op: free , state: count(n) , AS >
       OE <- sr(n, h, 0) .
  *** reset message
  eq < OS : Site | name: count, op: exec(reset, h, OE) , state: count(n) , AS > =
       < OS : Site | name: count , op: free , state: count(0) , AS >
       OE <- sr(sig, h, 0) .
endm

mod ORC-DF-SITE is
  inc ORC-INTERFACE .

  vars OS OE OE' : Oid .
  vars n i : Nat .
  vars h h' h'' : Handle .
  var AS : AttributeSet .
  var OL : OState .

  ************
  *** Eat Site
  ops SE1 SE2 SE3 SE4 : -> Oid [ctor] .
  ops eat : Nat -> SiteName [ctor] .
  *** eat message
  eq < OS : Site | name: eat(i), op: exec(nilA, h, OE) , AS > =
       < OS : Site | name: eat(i) , op: free , AS >
       OE <- sr(sig, h, 0) .

  **************
  *** Fork Sites
  ops SF1 SF2 SF3 SF4 : -> Oid [ctor] .
  op fork : Nat -> SiteName [ctor] .
  ---ops fork1 fork2 fork3 fork4 : -> SiteName [ctor] .
  ops obj : Oid Handle -> OState [ctor] .
  op cur : -> Handle [ctor] .

  *** get fork message
  eq < OS : Site | name: fork(i), op: exec(get, h, OE) , state: nil , AS > =
       < OS : Site | name: fork(i) , op: free , state: obj(OE , cur) , AS >
       OE <- sr(sig, h, 0) .

  eq < OS : Site | name: fork(i), op: exec(get, h, OE) , state: obj(OE', h') OL , AS > =
       < OS : Site | name: fork(i) , op: free , state: obj(OE, h) obj(OE', h') OL , AS > .

  *** put fork message
  eq < OS : Site | name: fork(i), op: exec(put, h, OE) ,
                   state: OL obj(OE',h') obj(OE, cur) , AS > =
       < OS : Site | name: fork(i) , op: free , state: OL obj(OE',cur) , AS >
       OE <- sr(sig, h, 0)
       OE' <- sr(sig, h', 0) .

  eq < OS : Site | name: fork(i), op: exec(put, h, OE) , state: obj(OE, cur) , AS > =
       < OS : Site | name: fork(i) , op: free , state: nil , AS >
       OE <- sr(sig, h, 0) .

endm

mod ORC-MEETING-SITE is
  inc ORC-INTERFACE .

  sort Status .
  ops ready pending scheduled canceled : -> Status .

  vars OS OE OE' : Oid .
  vars m n i t r : Nat . var p : NzNat .
```

```
var c c' : Const .  vars C C' T T' : ConstList .
vars h h' h'' : Handle .
var AS : AttributeSet .
var OL : OState .
var st : Status .

----op update : Nat -> Const .
op cancel : -> Const .

****************
*** Participant Site
op P : Nat -> Oid [ctor] .
op par : Nat -> SiteName [ctor] .
op sch : Const Status -> OState [ctor] .

--- get message when ready
eq < OS : Site | name: par(i), op: exec(get, h, OE) , state: sch(c, ready) , AS > =
     < OS : Site | name: par(i) , op: free , state: sch(c, pending) , AS >
     OE <- sr(c, h, rand) .

--- get message when ready, after being canceled
eq < OS : Site | name: par(i), op: exec(get, h, OE) , state: sch(c, canceled) , AS > =
     < OS : Site | name: par(i) , op: free , state: sch(c, pending) , AS >
     OE <- sr(c, h, rand) .

--- scheduled meeting time message
eq < OS : Site | name: par(i), op: exec(t, h, OE) ,
     state: sch(tup(C, t, C'), pending) , AS > =
     < OS : Site | name: par(i) , op: free , state: sch(tup(C, C'), scheduled) , AS >
     OE <- sr(sig, h, rand) .

--- cancel  message
eq < OS : Site | name: par(i), op: exec(cancel, h, OE) , state: sch(c, st) , AS > =
     < OS : Site | name: par(i) , op: free , state: sch(c, canceled) , AS >
     OE <- sr(sig, h, rand) .


****************
*** 'sMeetTime Site
op resolve : ConstList -> Const .
op resolve* : ConstList ConstList -> Const [comm] .

--- resolve meeting time
eq < OS : Site | name: 'sMeetTime, op: exec(tup(C), h, OE) , AS > =
     < OS : Site | name: 'sMeetTime , op: free , AS >
     OE <- sr(resolve(C), h, rand) .

eq resolve(tup(C), tup(C'), T)  = resolve(tup(resolve*(C, C')), T) .
eq resolve(tup(c, C)) = c .
eq resolve(tup(nilA)) = 0 .
```

```
 eq resolve(nilA)      = 0 .

 eq resolve*((c, C) , (c', C')) =
    if      c == c' then (c , resolve*(C, C'))
    else if c < c' then  resolve*(C, (c', C'))
    else                 resolve*((c, C), C') fi fi .
 eq resolve*(nilA , C) = nilA .

 *****************
 *** 'sRoomReserve Site
 op row : Nat Nat Const -> OState .

ceq < OS : Site | name: 'sRoomReserve, op: exec((t,n),  h, OE) ,
                        state: row(r, m, tup(C, t, C')) OL , AS > =
     < OS : Site | name: 'sRoomReserve, op: free , state: row(r, m, tup(C, C')) OL , AS >
     OE <- sr(r, h, rand)
       if n <= m .
 eq < OS : Site | name: 'sRoomReserve, op: exec((t,n),  h, OE) , state: OL , AS > =
     < OS : Site | name: 'sRoomReserve, op: free , state: OL , AS >
     OE <- sr(0, h, rand) [owise] .

 *****************
 *** 'sRoomCanceled Site
 op monitor : Nat Handle Oid -> OState .
 op cancel : Nat -> Const .

 --- request to monitor a room
 eq < OS : Site | name: 'sRoomCanceled , op: exec(r,  h, OE) , state: OL , AS > =
     < OS : Site | name: 'sRoomCanceled , op: free , state: monitor(r, h, OE) OL , AS > .

 --- emit a message if the room is canceled
 eq < OS : Site | name: 'sRoomCanceled , op: exec(cancel(r),  h, OE) ,
                  state: monitor(r, h, OE) OL , AS > =
     < OS : Site | name: 'sRoomCanceled , op: free , state: OL , AS >
     OE <- sr(sig, h, rand) .


 *****************
 *** configuration setup
 op meeting-sites : NzNat -> Configuration .
 op par-sites : NzNat -> Configuration .

 eq par-sites(2) =
    < S(21) : Site | name: par(1), op: free , state: sch(tup(3, 7, 8), ready) >
    < S(22) : Site | name: par(2), op: free , state: sch(tup(5, 6, 7, 8), ready) > .

 eq par-sites(3) =
    < S(21) : Site | name: par(1), op: free , state: sch(tup(3, 7, 8), ready) >
    < S(22) : Site | name: par(2), op: free , state: sch(tup(5, 6, 7, 8), ready) >
    < S(23) : Site | name: par(3), op: free , state: sch(tup(1, 2, 7), ready) > .
```

```
  eq  par-sites(4) =
      < S(21) : Site | name: par(1), op: free , state: sch(tup(3, 7, 8), ready) >
      < S(22) : Site | name: par(2), op: free , state: sch(tup(5, 6, 7, 8), ready) >
      < S(23) : Site | name: par(3), op: free , state: sch(tup(1, 2, 7), ready) >
      < S(24) : Site | name: par(4), op: free , state: sch(tup(3, 7, 9), ready) > .

  eq meeting-sites(p) =
      par-sites(p)
      < S(25) : Site | name: 'sMeetTime,     op: free , state: nil >
      < S(26) : Site | name: 'sRoomReserve,  op: free , state:
                 row(1120, 5, tup(1, 2, 5, 7, 8, 9)) row(1120, 3, tup(4, 8, 9)) >
      < S(27) : Site | name: 'sRoomCanceled, op: free , state: nil > .
endm

mod ORC-AUCTION-SITE is
  inc ORC-INTERFACE .

  sort Item ItemList Req ReqList .
  subsort Item < ItemList .
  subsort Req < ReqList .

  vars OS OE OE' : Oid .
  vars id m n n' i t r : Nat . var p : NzNat .
  var c c' : Const .  vars C C' T T' : ConstList .
  vars h h' h'' : Handle .
  var AS : AttributeSet .
  var OL : OState .
  var M M' : SiteName .
  var RQ : ReqList . var IT : ItemList .

  ops seller bidder : Nat -> SiteName .
  ops auction maxBid none : -> SiteName .

  ops postNext getNext post nextBid won : -> Const .

  op iNil : -> ItemList .
  op _,_ : ItemList ItemList -> ItemList [assoc id: iNil] .
  op rNil : -> ReqList .
  op _,_ : ReqList ReqList -> ReqList [assoc id: rNil] .

  ----op bNil : -> BidList .
  ----op _,_ : BidList BidList -> BidList [assoc id: bNil] .
  ----op wNil : -> WonList .
  ----op _,_ : WonList WonList -> ReqList [assoc id: rNil] .

  op itms : ItemList -> OState .
  op reqs : ReqList -> OState .

  op item : Nat Nat Nat -> Item .
  op req : Oid Handle -> Req .

  op witem : Nat Nat -> OState .
  op bid : Nat Nat -> OState .

  **** seller site
  ****************

  eq < OS : Site | name: seller(i) , op: exec(postNext, h,  OE) ,
                   state: itms(item(id, t, m), IT) , AS > =
       < OS : Site | name: seller(i) , op: free , state: itms(IT)  , AS >
       OE <- sr(tup(id, t, m), h, rand) .


  **** auction site
  *****************

  --- post message
  eq < OS : Site | name: auction , op: exec((post, tup(id, t, m)), h,  OE) ,
                  state: reqs(RQ) itms(IT) , AS > =
       < OS : Site | name: auction , op: free , state: reqs(RQ)
                                                      itms(IT, item(id, t, m)) , AS >
       OE <- sr(sig, h, rand) .

  --- get message
  eq < OS : Site | name: auction , op: exec(getNext, h,  OE) ,
                   state: reqs(RQ) itms(IT) , AS > =
       < OS : Site | name: auction , op: free ,
                     state: reqs(RQ, req(OE, h)) itms(IT) , AS > .


  --- servicing a request
  eq < OS : Site | name: auction , op: free ,
                   state: reqs(req(OE, h) , RQ) itms(item(id, t, m), IT) , AS > =
       < OS : Site | name: auction , op: free , state: reqs(RQ) itms(IT) , AS >
       OE <- sr(tup(id, t, m), h, rand) .


  **** bidder site
  ****************

  --- won message
  eq < OS : Site | name: bidder(i) , op: exec((won, id, m), h,  OE) , state: OL , AS > =
       < OS : Site | name: bidder(i) , op: free , state: OL witem(id, m)  , AS >
       OE <- sr(sig, h, rand) .

  --- bid message
  eq < OS : Site | name: bidder(i) , op: exec((nextBid, id, m), h,  OE) ,
                   state: bid(id, n) OL , AS > =
       < OS : Site | name: bidder(i) , op: free , state: bid(id, s(m + i)) OL  , AS >
```

```
       OE <- sr(tup((s(m + i)), bidder(i)), h, rand) .

  eq < OS : Site | name: bidder(i) , op: exec((nextBid, id, m), h,  OE) , state: OL , AS > =
       < OS : Site | name: bidder(i) , op: free , state: bid(id, s(m + i)) OL  , AS >
       OE <- sr(tup((s(m + i)), bidder(i)), h, rand) [owise] .

  **** maxBid site
  ****************

  op mb : Const ConstList -> Const .

  ---  computing the largest bid
  eq < OS : Site | name: maxBid , op: exec(tup(c, C), h,  OE) , state: nil  , AS > =
       < OS : Site | name: maxBid , op: free , state: nil  , AS >
       OE <- sr(mb(tup(0, none), (c, C)), h, rand) .

  eq mb(tup(n, M), (tup(n', M') , c, C)) =
       if (n' > n) then mb(tup(n', M'), (c, C))
       else           mb(tup(n, M), (c, C))  fi .
  eq mb(tup(n, M), tup(n', M')) =
       if (n' > n) then tup(n', M')
       else           tup(n, M) fi .

  op auction-sites : -> Configuration .
  eq auction-sites =
    < S(20) : Site | name: seller(0) , op: free ,
                     state: itms(item(1910, 5, 500), item(1720, 8, 700)) >
    < S(21) : Site | name: bidder(0) , op: free , state: nil  >
    < S(22) : Site | name: bidder(1) , op: free , state: nil  >
    < S(23) : Site | name: bidder(2) , op: free , state: nil  >
    < S(24) : Site | name: auction , op: free , state: reqs(rNil) itms(iNil) >
    < S(25) : Site | name: maxBid , op: free , state: nil > .
endm
```

A.2.5 LTL Model Checking Modules for auction

```
in orc-objexamples.maude
in ../../maude/model-checker.maude

mod AU-PREDS is
  pr EXAMPLES .
  inc SATISFACTION .

  subsort System < State .

  sort Pair .
  op [_,_] : Const Const -> Pair .
  ops first second : Pair -> Const .

  ops hasBid sold maxbid maxbidder max conflict : Nat -> Prop .
```

```
 ops mxbd mxbdr : Nat Configuration -> Const .
 op mxbd* : Nat Nat SiteName Configuration -> Pair .

 var S : System . var CF : Configuration .
 vars id t i j m m' : Nat . vars OL OL' OL1 OL2 OL3 OL4 : OState .
 vars x x' : Var .
 vars f f' f'' : Expr .
 var OE : Oid .
 vars AS AS' : AttributeSet .
 var c c' : Const .
 var B : SiteName .

 eq first([c,c']) = c .
 eq second([c,c']) = c' .

 eq { < OE : Site | name: bidder(i) , state: OL witem(id, m) OL'  , AS > CF }
        |= sold(id) = true .

 eq { < OE : Site | name: bidder(i) , state: OL bid(id, m) OL'  , AS > CF }
        |= hasBid(id) = true .

 eq { < OE : Site | name: bidder(i) , state: OL1 witem(id, m) OL2  , AS >
      < OE : Site | name: bidder(j) , state: OL3 witem(id, m') OL4 , AS' > CF } |=
       conflict(id) = true .

 eq { < OE : Site | name: bidder(i) , state: OL witem(id, m) OL'  , AS > CF }
    |= maxbid(id)
   = m >= mxbd(id, < OE : Site | name: bidder(i) ,
                                  state: OL witem(id, m) OL'  , AS > CF) .

 eq { < OE : Site | name: bidder(i) , state: OL witem(id, m) OL'  , AS > CF }
    |= maxbidder(id)
   = bidder(i) == mxbdr(id, < OE : Site | name: bidder(i) ,
                                      state: OL witem(id, m) OL'  , AS > CF) .

ceq { < OE : Site | name: bidder(i) , state: OL witem(id, m) OL'  , AS > CF }
    |= max(id)
    = true if
  m >= mxbd(id, < OE : Site | name: bidder(i) ,
                                 state: OL witem(id, m) OL'  , AS > CF) /\
  bidder(i) == mxbdr(id, < OE : Site | name: bidder(i) ,
                                      state: OL witem(id, m) OL'  , AS > CF) .

 eq mxbd(id, < OE : Site | name: bidder(i) ,
                                           state: OL bid(id, m) OL'  , AS > CF)
    = first(mxbd*(id, m, bidder(i), CF)) .

 eq mxbdr(id, < OE : Site | name: bidder(i) ,
                                    state: OL bid(id, m) OL'  , AS > CF)
```

```
     = second(mxbd*(id, m, bidder(i), CF)) .

  eq mxbd*(id, m, B, < OE : Site | name: bidder(i) ,
                                   state: OL bid(id, m') OL'  , AS > CF)
     = if m' > m then mxbd*(id, m', bidder(i), CF)
       else           mxbd*(id, m, B, CF)
       fi .
  eq mxbd*(id, m, B, CF) = [m, B] [owise] .
endm

mod AU-CHECK is
  pr AU-PREDS .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
  op init : -> System .
  ops commit winbid winbidder win uniqueWinner : Nat -> Formula .
  ops commitAll winbidAll winbidderAll uniqueWinnerAll winAll : -> Formula .

  var i n id : Nat .

  eq commit(id) = hasBid(id) -> <> sold(id) .
  eq commitAll = [] (commit(1910) /\ commit(1720)) .

  eq winbid(id) = sold(id) -> maxbid(id) .
  eq winbidAll = [] (winbid(1910) /\ winbid(1720)) .

  eq winbidder(id) = sold(id) -> maxbidder(id) .
  eq winbidderAll = [] (winbidder(1910) /\ winbidder(1720)) .

  eq win(id) = sold(id) -> max(id) .
  eq winAll = [] (win(1910) /\ win(1720)) .

  eq uniqueWinnerAll = [] ~ (conflict(1910) \/ conflict(1720)) .

  eq init =
    $ [ E(1) : Posting ; PostingExpr ]
      [ E(2) : Bidding ; Bids ; Collect ; BiddingExpr ]
      auction-sites $ .
endm
```

B Timed SOS-Based Rewriting Transformation Methodology

We describe and formally prove the correctness of the *eagerEnabled* predicate transformation that was used in Section 5.4. This is done by describing a general method, of which the transformation in Section 5.4 is an instance, to embed a discrete time domain into the SOS specifications of a language $\mathcal{L}$ given as a rewrite theory $\mathcal{R}_\mathcal{L}$ in rewriting logic. We specify constraints on $\mathcal{R}_\mathcal{L}$ under which the methodology yields a theory $\mathcal{R}'_{\mathcal{L}}$ with a correct behavior, and then prove its correctness.

B.1 The Transformation Methodology

Suppose that $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}}, \phi_{\mathcal{L}})$ is a rewrite theory representing the (non-timed) SOS definitions of a language $\mathcal{L}$ (typically obtained by the modular methodology explained in [4]). In accordance with [4], we assume that a configuration in $\mathcal{R}_{\mathcal{L}}$ is given by a pair $\langle E, R\rangle$ with E a program expression and R a record consisting of fields each of which maintains a piece of state information, such as a clock or an environment, required to compute E. A field is itself a pair $(i : \mathit{comp})$, with i an index identifying the field and comp a component holding the data relevant to the field. Fields are grouped into a record using the associative and commutative "|" operator. We also assume that one-step rewrites, corresponding to small-step SOS specifications, are implemented as rules of the form $\{E, R\} \rightarrow [E, R]$ if *cond*.

Given $\mathcal{R}_{\mathcal{L}}$, we first extend the signature $\Sigma_{\mathcal{L}}$ with a declaration of a (partial) predicate *eagerEnabled* as follows,

$$\mathit{eagerEnabled} \ : \ \mathit{Conf} \ \rightarrow \ \texttt{[Bool] [frozen]}$$

with *Conf* the sort of a configuration of $\mathcal{L}$. The predicate, as is, serves as a means by which one can tell whether a rule in $R_{\mathcal{L}}$ is applicable to an instance of a configuration in $\mathcal{L}$. In the next section we will prove a stronger assertion that, under certain assumptions, the predicate tells us exactly when a rule in $R_{\mathcal{L}}$ can be taken on a configuration in $\mathcal{L}$. The reader might note that the predicate is declared as a frozen operator to avoid useless rewrites in the configuration while the predicate is computed.

$E_{\mathcal{L}}$ is then extended with equations defining the *eagerEnabled* predicate as follows. For each rule $r : \{E, R\} \rightarrow [E', R']$ if $C \wedge \bigwedge_{i=1}^{n} \{E_i, R_i\} \rightarrow [E'_i, R'_i]$ in $R_{\mathcal{L}}$, with C a possibly empty conjunction of equational conditions (memberships and/or equations) and $n \geq 0$ [13] , we introduce an equation

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

Finally, a new "clock" field of the form $\mathit{clk} : \mathit{clock}(c)$ is added to the fields of the record, and a clock tick rule of the form

$$\begin{gathered}\{E, R \mid \mathit{clk} : \mathit{clock}(c)\} \rightarrow \{E', R' \mid \mathit{clk} : \mathit{clock}(c+1)\} \\ \text{if } \mathit{eagerEnabled}(E, R \mid \mathit{clk} : \mathit{clock}(c)) \neq \mathit{true}\end{gathered}$$

is added to $R_{\mathcal{L}}$. The tick rule advances the clock by one time unit and propagates the effect of the lapse of time on the other components of R. Such rule is usually

[13]Here we are implicitly assuming that every rule in $R_{\mathcal{L}}$ is of this form. We will elaborate on that and justify it in the next section.

referred to as a *lazy* or a *tick* rule, whereas rules in $R_{\mathcal{L}}$ are called *eager* or *instantaneous* rules. We refer to the resulting theory as $\mathcal{R}'_{\mathcal{L}} = (\Sigma'_{\mathcal{L}}, E'_{\mathcal{L}}, R'_{\mathcal{L}}, \phi'_{\mathcal{L}})$.

For this approach to be sound, however, we make the following, quite reasonable assumptions about the rewrite theory $\mathcal{R}_{\mathcal{L}}$. A more formal description of these assumptions is deferred to the next section.

1. $E_{\mathcal{L}}$ is confluent and terminating, and $R_{\mathcal{L}}$ is coherent with respect to $E_{\mathcal{L}}$.

2. All rules in $R_{\mathcal{L}}$ are structural.

3. Rules in $R_{\mathcal{L}}$ are either unconditional (axioms), conditional with equational conditions, or conditional with equations and rewrites in the condition. If the condition of a rule consists of equations and rewrites, then all newly introduced variables are local to the equations or rewrites which introduced them. In other words, no chaining of newly introduced variables is allowed in any rule of $R_{\mathcal{L}}$. The idea is to ensure that new variables in a conjunct are not referenced in other conjuncts, as this would result in ill-formed conditional equations of the *eagerEnable* predicate.

4. If a configuration could evolve in different ways, then all such ways are accounted for in $R_{\mathcal{L}}$.

For almost all practical purposes, the above-mentioned constraints are not at all restrictive. In fact, when giving SOS definitions of the semantics of almost any language, the corresponding rewrite theory is highly likely to automatically satisfy these conditions. This can easily be seen by recalling that a semantic rule in the SOS definitions of almost any language is structural (although this is not a requirement of SOS itself), has premises that are independent of each other, and preserves a configuration structure. Moreover, even when the definitions do not satisfy some of these conditions, they can often be easily transformed into equivalent definitions satisfying them. This underscores the fact that we are still dealing with a fairly large class of SOS specifications.

B.2 Correctness of the Methodology

In this section we prove some properties about $\mathcal{R}'_{\mathcal{L}}$. More importantly, we show why a system implemented using this methodology behaves as described above.

B.2.1 Basic Definitions and Notation

We begin by introducing some basic definitions. We say that a term t is of the form (or has the form, or is an instance of) t' if there exists a substitution θ such that $\theta(t') = t$. This notion is extended to equations, memberships and rewrite rules in the obvious way. Moreover, we define equality among terms of the same sort as syntactic equality modulo variable renaming. We also say that a term t is a strict subterm of t' if t is a subterm of t' and $t \neq t'$. For a rewrite rule $r : t \rightarrow t'$ if *cond*, we let $lhs(r) = t$, $rhs(r) = t'$, $eqc(r)$ be the equational part of the condition *cond*, and $rwc_i(r)$ be the ith rewrite condition of *cond*. For instance, if $r : f(a) \to f'(b)$ if $g(a) = t \wedge h(a) \to h'(c) \wedge p(c) : s \wedge q(c) \to q'(b)$, then $eqc(r) = g(a) = t \wedge p(c) : s$, $rwc_1(r) = h(a) \to h'(c)$, and $rwc_2(r) = q(c) \to q'(b)$.

We let E and R (and their decorated variants) range over terms of sort *Expr* and *Record*, respectively, and W, X, Y, Z and their variants range over sets of variables. We denote by R_E the set of eager rules in $\mathcal{R}$, and by P_E the corresponding set of equations defining the *eagerEnabled* predicate. We also let $R_E^n \subseteq R_E$ be the set of eager rules with exactly n rewrite conditions. Finally, we call a configuration of the form $\{E, R\}$, $[E, R]$, or $\langle E, R \rangle$ a valid configuration if R is a record having all the necessary fields required by the semantics of the language.

In addition, the following construction will prove useful later. We define a partition $\mathcal{P}$ on R_E based on the left-hand sides of the rules and the left-hand sides of their rewrite conditions, such that two rules are in the same partition if and only if they share the same left-hand side and have fixed number n of rewrite conditions that (under an appropriate re-ordering) share the same corresponding left-hand sides. More precisely, a partition $P_{t,t_1,\cdots,t_n}$ is defined as follows,

$$\begin{aligned} P_{t,t_1,\cdots,t_n} \quad = \quad & \{r \in R_E^n \mid lhs(r) = t \text{ and if } n > 0 \text{ then there exists } \pi \text{ such that} \\ & lhs(rwc_i(r)) = t_{\pi(i)} \text{ for all } i, 1 \leq i \leq n\} \end{aligned}$$

with π a permutation of $[1 \ldots n]$. We write $\mathcal{P}_{R_E}$ for the set of all such partitions. Moreover, if $X = \bigcup_{i=1}^{n} vars(t_i)$, then we denote by $\Theta_{t,t_1,\cdots,t_n}$ the set of substitutions on X such that if $\theta \in \Theta_{t,t_1,\cdots,t_n}$ then for each i, there exists $r_i \in R_E$ and a substitution ρ_i, such that $\theta(t_i) = \rho_i(lhs(r_i))$. We call θ a *simultaneous substitution* on the terms t_i.

B.2.2 Constraints

Beside the confluence and termination of $E_{\mathcal{L}}$, and the coherence of $R_{\mathcal{L}}$ with respect to $E_{\mathcal{L}}$, we assume that for any $r \in R_E$, r is of one of two forms (which we shall refer to throughout the rest of the section), either

$$\{E(W), R(X) \mid \mathtt{R}\} \;\to\; [E'(Y), R'(Z) \mid \mathtt{R}] \text{ if } C(Y, Z) \tag{5}$$

or

$$\begin{gathered} \{E(W), R(X) \mid \mathtt{R}\} \;\to\; [E'(Y), R'(Z) \mid \mathtt{R}] \\ \text{if } C(Y_c, Z_c) \wedge \; \bigwedge_{i=1}^{n} \{E_i(W), R_i(X) \mid \mathtt{R}\} \to [E_i'(Y_i), R_i'(Z_i) \mid \mathtt{R}] \end{gathered} \tag{6}$$

where in both cases C is a possibly empty conjunction of equational conditions, and $\mathtt{R}$ is record variable standing for "the rest of the record"[14], and in the second case $n > 0$, such that the following local properties hold.

[14]The record variable R (along with the structural axioms of the "|" operator) is used to implement the *record inheritance* modularity technique explained in [4] in Maude. Record inheritance is important in our construction since it allows us to add the clock field to the record structure of a configuration without requiring any change to be made to the instantaneous rules to preserve their applicability.

1. If r is of the form (6), then E_i is a strict subterm of E, for all $i \leq n$ (being structural). Moreover, $\{Y_c, Z_c\} \subseteq \{Y, Z\}$, and for all i, if $v \in \{Y_i, Z_i\} - \{W, X\}$ then for all $j \neq i$, $v \notin \{Y_j, Z_j\} - \{W, X\}$, and, finally, for all i, $\{Y_i, Z_i\} \subseteq \{Y, Z\}$

2. In any rule r, for every field $f = (i : \mathit{comp})$ for some index i and component comp, if $R = f \mid \bar{R}$ for some record $\bar{R}$, then there exists a field $f' = (i : \mathit{comp}')$ with some component comp' and there exists a record $\bar{R}'$ such that $R' = f' \mid \bar{R}'$ (preservation of configuration).

In addition to the local constraints given above, two global constraints must be met by R_E, as follows. If $P_{t,t_1,\cdots,t_n} \in \mathcal{P}_{R_E}$ and if we denote the jth rule in $P_{t,t_1,\cdots,t_n}$ by $t \to t_{(j)}$ if $c_{(j)} \wedge \bigwedge_{i=1}^{n} t_i \to t'_{i_{(j)}}$, then

1. if $r, r' \in P_{t,t_1,\cdots,t_n}$ then $eqc(r) = eqc(r')$, and

2. for each simultaneous substitution $\theta \in \Theta_{t,t_1,\cdots,t_n}$, there exists j such that for all i with $1 \leq i \leq n$, there exists $r_i \in R_E^n$ and a substitution ρ_i such that $\rho_i(lhs(r_i)) = \theta(t_i)$ and $\rho_i(rhs(r_i))$ is an instance of $\theta(t'_{i_{(j)}})$)).

The above local and global constraints give sufficient conditions under which the predicate *eagerEnabled*, when introduced to $\mathcal{R}_\mathcal{L}$, is well-defined. They also guarantee that the theory $\mathcal{R}_\mathcal{L}$ represents a well-defined system, and that R_E specifies all the ways in which it could evolve.

Now considering $\mathcal{R}'_\mathcal{L}$, the discretely timed theory that one would obtain by applying the transformation described above on $\mathcal{R}_\mathcal{L}$, we observe that the constraints above imply similar properties for the *eagerEnabled* predicate. More specifically, each $p \in P_E$ (the set of equations defining the *eagerEnabled* predicate in $\mathcal{R}_\mathcal{L}{}'$) has one of two forms, namely

$$\mathit{eagerEnabled}(\{E(W), R(X)\}) = \mathit{true} \text{ if } C(Y, Z) \tag{7}$$

or

$$\begin{aligned} &\mathit{eagerEnabled}(\{E(W), R(X)\}) = \mathit{true} \\ &\text{if } C(Y_c, Z_c) \wedge \ \textstyle\bigwedge_{i=1}^{n} \mathit{eagerEnabled}(\{E_i(W), R_i(X)\}) = \mathit{true}. \end{aligned} \tag{8}$$

Moreover, if p is of the form (8), E_i is a strict subterm of E, for all $i \leq n$. We also note that by construction, for each rule r there is exactly one *eagerEnabled* predicate associated with it. In fact, one can easily see that this association is actually a bijection between P_E and P_{R_E}. This bijection, however, is purely conceptual and has no real significance in our methodology. It merely makes thinking about the following proofs easier and perhaps more modular.

B.2.3 Executability Requirements of $\mathcal{R}_\mathcal{L}{}'$

In the original theory $\mathcal{R}_\mathcal{L}$, executability requirements, namely confluence and termination of $E_\mathcal{L}$, and coherence of $R_\mathcal{L}$ with respect to $E_\mathcal{L}$, are assumed to be satisfied. Here, we show that such requirements are also satisfied in the theory $\mathcal{R}'_{\mathcal{L}} = (\Sigma'_{\mathcal{L}}, E'_{\mathcal{L}}, R'_{\mathcal{L}}, \phi'_{\mathcal{L}})$. We first observe that we may assume without loss of generality that the kind `[Bool]` is kept intact with only one sort, namely `Bool`, and two constructors, `true` and `false`, and that the *eagerEnabled* predicate is the only operator of this kind. We also note that, by construction, no term of the kind `[Bool]` may occur as a subterm of a term of another kind.

Lemma 6. *$E'_{\mathcal{L}}$ is confluent and terminating.*

Proof. Let $t \in T_{\Sigma'_{\mathcal{L}},k}(X)$ for some kind k. Suppose $r \overset{*}{\leftarrow} t \overset{*}{\rightarrow} s$ for some terms $r, s \in T_{\Sigma'_{\mathcal{L}},k}(X)$. If $k \neq [\texttt{Bool}]$, then $T_{\Sigma'_{\mathcal{L}},k}(X) = T_{\Sigma_{\mathcal{L}},k}(X)$ and, by the confluence of $E_{\mathcal{L}}$, there exists a term t' such that $r \overset{*}{\rightarrow} t' \overset{*}{\leftarrow} s$. Otherwise, if $k = [\texttt{Bool}]$, then t is either *true*, *false*, or of the form *eagerEnabled*$(\mathcal{C})$ for some configuration $\mathcal{C} \in T_{\Sigma'_{\mathcal{L}},[\texttt{Configuration}]}(X) = T_{\Sigma_{\mathcal{L}},[\texttt{Configuration}]}(X)$. Since *true* and *false* are in canonical form, we only need to consider the third case, where, either $r = s = \texttt{true}$ or $r =$ *eagerEnabled*$(\mathcal{C}_1)$ and $s =$ *eagerEnabled*$(\mathcal{C}_2)$ for some configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ such that $\mathcal{C}_1 \overset{*}{\leftarrow} \mathcal{C} \overset{*}{\rightarrow} \mathcal{C}_2$. By the confluence of $E_{\mathcal{L}}$, we know that there exists a configuration $\mathcal{C}'$ such that $\mathcal{C}_1 \overset{*}{\rightarrow} \mathcal{C}' \overset{*}{\leftarrow} \mathcal{C}_2$ and thus $r \overset{*}{\rightarrow}$ *eagerEnabled*$(\mathcal{C}') \overset{*}{\leftarrow} s$. Therefore, $E'_{\mathcal{L}}$ is confluent.

For termination of $E'_{\mathcal{L}}$, suppose $E'_{\mathcal{L}}$ is not terminating. Then, there exists an infinite sequence of rewrites starting at some term $t \in T_{\Sigma'_{\mathcal{L}},k}(X)$. Since $E_{\mathcal{L}}$ is assumed terminating, t can only be of the kind `[Bool]`. This can only happen if t is of the form *eagerEnabled*$(\mathcal{C})$ for some configuration $\mathcal{C} \in T_{\Sigma_{\mathcal{L}},[\texttt{Configuration}]}(X)$. Since $E_{\mathcal{L}}$ is terminating, there is no infinite sequence of rewrites starting at $\mathcal{C}$, and thus the only way t may not terminate is by rewriting at the top operator of t, namely the *eagerEnabled* operator. However, if *eagerEnabled*$(\mathcal{C}) \overset{1}{\rightarrow} t'$ at the top, then $t' = \texttt{true}$, which implies that sequences of rewrites starting at t are all finite, contradicting the assumption. Therefore, $E'_{\mathcal{L}}$ is terminating. □

Lemma 7. *$R'_{\mathcal{L}}$ is coherent with respect to $E'_{\mathcal{L}}$.*

Proof. let $t \in T_{\Sigma_{\mathcal{L}},k}(X)$ for some kind k. If $k \neq [\texttt{Bool}]$, then there is nothing to prove, by assumption. So suppose $k = [\texttt{Bool}]$. Then, $t =$ *eagerEnabled*$(\mathcal{C})$ for some configuration $\mathcal{C}$. Since *eagerEnabled* is declared frozen at its argument $\mathcal{C}$, and no rule rewrites a term of kind `[Bool]` in $R'_{\mathcal{L}}$, it follows that $R'_{\mathcal{L}}$ is coherent with respect to $E'_{\mathcal{L}}$. □

B.2.4 Correctness

We now prove the main result of this section. We show that in $\mathcal{R}'_{\mathcal{L}}$, a tick rule will never be applied to a state at which some eager rule can still be taken.

Theorem 2. *For all $E_t(W_t)$ and $R_t(X_t)$ such that $\{E_t(W_t), R_t(X_t)\}$ is a valid configuration, $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \text{eagerEnabled}(\{E_t(W_t), R_t(X_t)\}) = \text{true}$ if and only if there exists a valid configuration $\mathcal{C}$ such that*

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \{E_t(W_t), R_t(X_t)\} \rightarrow \mathcal{C}.$$

Proof. ($\Longleftarrow$) Assuming there exists a valid configuration $\mathcal{C}$ such that

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \{E_t(W_t), R_t(X_t)\} \rightarrow \mathcal{C},$$

we show that $(\forall W_t, X_t)$ *eagerEnabled*$(\{E_t(W_t), R_t(X_t)\}) = \mathit{true}$ is provable from $R'_{\mathcal{L}}$ by induction on the number n of Nested Replacement inference rule applications in a proof of $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \{E_t(W_t), R_t(X_t)\} \rightarrow \mathcal{C}$.

Base Case. When $n = 0$, there exists a rewrite rule $r \in R_E$ of the form (5), with an associated equation of the form (7), such that $\theta(\{E(W), R(X)\}) = \{E_t(W_t), R_t(X_t)\}$ and $\mathcal{C} = \theta([E'(Y), R'(Z)])$, for some substitution θ. Therefore, $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \theta(C(Y, Z)) = \mathit{true}$, which implies

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \mathit{eagerEnabled}(\theta(\{E(W), R(X)\})) = \mathit{true},$$

as asserted.

Inductive Step. Suppose for $k = n - 1$, where $n > 0$, the proposition holds. Since $n > 0$, there exists a rewrite rule $r \in R_E$ of the form (6), with an associated equation of the form (8), with $\theta(\{E(W), R(X)\}) = \{E_t(W_t), R_t(X_t)\}$ and $\mathcal{C} = \theta([E'(Y), R'(Z)])$, for some substitution θ. This implies, by the Nested Replacement inference rule that, $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \ \theta(C(Y_c, Z_c)) = \mathit{true}$, and for all i,

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \theta(\{E_i(W), R_i(X)\}) \rightarrow \theta([E'_i(Y_i), R'_i(Z_i)]).$$

By the induction hypothesis, this implies for all i,

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \mathit{eagerEnabled}(\theta(\{E_i(W), R_i(X)\})) = \mathit{true},$$

which implies $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)$ *eagerEnabled*$(\theta(\{E(W), R(X)\})) = \mathit{true}$. This concludes the proof of this direction of the implication.

($\Longrightarrow$) Suppose $R'_{\mathcal{L}}$ proves $(\forall W_t, X_t)$ *eagerEnabled*$(\{E_t(W_t), R_t(X_t)\}) = \mathit{true}$. We show that there exists a valid configuration $\mathcal{C}$ such that

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \{E_t(W_t), R_t(X_t)\} \rightarrow \mathcal{C}$$

by induction on the number n of applications of the modus ponens inference rule in the equational deduction of *eagerEnabled*$(\{E_t(W_t), R_t(X_t)\}) = \mathit{true}$.

Base Case. When $n = 1$, there exists an equation of the form (7) with an associated rewrite rule of the form (5), with $\theta(\{E(W), R(X)\}) = \{E_t(W_t), R_t(X_t)\}$. Therefore, $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \theta(C(Y_c, Z_c)) = \mathit{true}$, which implies by the Replacement rule, that $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \theta(\{E(W), R(X)\}) \rightarrow \theta([E'(W), R'(X)])$, and hence $\mathcal{C} = \theta([E'(W), R'(X)])$.

Inductive Step. Suppose the proposition is true for all $k < n$. Since $n > 0$, there exists an equation of the form (8), with at least one associated rewrite rule of the form (6), such that $\theta(\{E(W), R(X)\}) = \{E_t(W_t), R_t(X_t)\}$. This implies that $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \theta(C(Y_c, Z_c)) = \mathit{true}$, and for all i, $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)$ *eagerEnabled*$(\theta(\{E_i(W), R_i(X)\})) = \mathit{true}$. By the induction hypothesis, this implies for all i, there exists a valid configuration $\mathcal{C}_i$ such that $R'_{\mathcal{L}} \vdash (\forall W_t, X_t)$ $\theta(\{E_i(W), R_i(X)\}) \rightarrow \mathcal{C}_i$.

Now since θ is a simultaneous substitution on $\{E_i(W), R_i(X)\}$, then there exists a rule $r_j \in P_{\{E,R\},\{E_1,R_1\},\ldots,\{E_n,R_n\}}$ such that for all i, there exists a substitution ρ such that $\mathcal{C}_i = \rho(\theta([E'_i(W), R'_i(X)]_{(j)}))$ and $\rho|_{W\cup X\cup Y_c \cup Z_c} = id$. Thus, for all i,

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \theta(\{E_i(W), R_i(X)\}) \to \rho(\theta([E'_i(W), R'_i(X)]_{(j)})),$$

which implies that

$$R'_{\mathcal{L}} \vdash (\forall W_t, X_t)\ \theta(\{E(W), R(X)\}) \to \rho(\theta([E'(W), R'(X)]_{(j)})),$$

and therefore $\mathcal{C} = \rho(\theta([E'(W), R'(X)]_{(j)}))$. This concludes the proof. $\square$

C Proofs of the Results of Section 7

In what follows, we refer to the theories $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ respectively by $\mathcal{R}^{s}_{Orc}$ and $\mathcal{R}^{r}_{Orc}$ for brevity.

Proof of Lemma 2

Proof. Suppose $\langle f, tr : t \mid r\rangle \to_{\mathcal{R}^s_{Orc}} \langle f', tr : t' \mid r'\rangle$. Then, there exists a rewrite rule of the form $\langle f_w; tr : t_w \mid r_w\rangle \to_{\mathcal{R}^s_{Orc}} \langle f'_w; tr : t'_w \mid r'_w\rangle$ with t_w a variable of sort EventList and t'_w being either the variable t_w or the term $t_w.L$ for some variable-free term L of sort Event, and a substitution θ such that $\theta(t_w) = t$ and $\theta(t'_w) = t'$. Thus, if $t'_w = t_w$, then $t' = \theta(t'_w) = \theta(t_w) = t$. Otherwise, if $t'_w = t_w.L$, then $t' = \theta(t'_w) = \theta(t_w.L) = \theta(t_w).L = t.L$. Consequently, for any EventList s, we can choose another substitution α_s such that $\alpha_s(t_w) = s.t$ and equal to θ otherwise. Now, if $t'_w = t_w$, then $\alpha_s(t'_w) = s.t = s.t'$. Otherwise, $\alpha_s(t'_w) = (s.t).L$ which by associativity of the . operator is equal to $s.(t.L) = s.t'$.

A similar argument applied to the rules and equations of $\mathcal{R}^r_{Orc}$ proves the property holds in $\mathcal{R}^r_{Orc}$ as well. $\square$

Proof of Lemma 3

Proof. (1) Suppose $L = M\langle C, h|m\rangle$. Then $f \to_{\mathcal{R}^r_{Orc}} sc^{\uparrow}(\hat{f}, M, C)$, with $\hat{f} = f[p \leftarrow \gamma]$ for some position p in f. By the congruence rule, $f \mid g \to_{\mathcal{R}^r_{Orc}} sc^{\uparrow}(\hat{f}, M, C) \mid g = sc^{\uparrow}(\hat{f} \mid g, M, C)$, which implies

$$\begin{aligned}
\langle f \mid g, tr : t \mid r\rangle &\to_{\mathcal{R}^r_{Orc}} \langle sc^{\uparrow}(\hat{f} \mid g, M, C), tr : t \mid r\rangle \\
&= \langle sc^{\downarrow}(\hat{f} \mid g, h), tr : t.L \mid r'\rangle \text{ with } h \text{ the current handle given by } r \\
&= \langle sc^{\downarrow}(\hat{f}, h) \mid sc^{\downarrow}(g, h), tr : t.L \mid r'\rangle \\
&= \langle f' \mid g, tr : t.L \mid r'\rangle
\end{aligned}$$

For the other direction, we have by the SiteCall rule, $f \mid g \to_{\mathcal{R}^r_{Orc}} sc^{\uparrow}(\hat{f} \mid g, M, C) = sc^{\uparrow}(\hat{f}, M, C) \mid g$, which by congruence implies $f \to_{\mathcal{R}^r_{Orc}} sc^{\uparrow}(\hat{f}, M, C)$ and thus,

$$\begin{aligned}
&\langle f, tr : t \mid r\rangle \to_{\mathcal{R}^r_{Orc}} \langle sc^{\uparrow}(\hat{f}, M, C), tr : t \mid r\rangle \\
&\quad = \langle sc^{\downarrow}(\hat{f}, h), tr : t.L \mid r'\rangle \text{ with } h \text{ the current handle given by } r \\
&\quad = \langle f', tr : t.L \mid r'\rangle
\end{aligned}$$

The cases for L a publishing of event and a τ event are similar. If $L = c?h|m$, then by the SiteRet rule, h appears in f, $active(f) \neq true$, and $f' = f[p \leftarrow !c]$. Since $active(g) \neq true$, we have

$$\begin{aligned}
&\langle f \mid g, tr : t \mid r\rangle \to_{\mathcal{R}^r_{Orc}} \langle sr(f \mid g, c, h), tr : t.L \mid r'\rangle \\
&\quad = \langle sr(f, c, h) \mid sr(g, c, h), tr : t.L \mid r'\rangle \\
&\quad = \langle f' \mid g, tr : t.L \mid r'\rangle
\end{aligned}$$

For the other direction, the SiteRet rule implies $active(f) \neq true$ and $active(g) \neq true$, and that h appears in $f \mid g$. However, since $sr(g, c, h) = g$, g is not changed, and thus h can only appear in f. Therefore, an application of the SiteRet rule and the equations for sr conclude the proof.

(2) ($\Longrightarrow$) we have $\langle f, tr : t \mid r\rangle \to_{\mathcal{R}^r_{Orc}} \langle pub(f', c), tr : t.(!c|m) \mid r'\rangle$, with $f' = f[p \leftarrow 0]$, for some position p in f. By congruence, this implies $f > x > g \to_{\mathcal{R}^r_{Orc}} pub(f', c) > x > g = pub^{\tau}(f' > x > g \mid g\{c/x\})$. Therefore,

$$\begin{aligned}
&\langle f > x > g, tr : t \mid r\rangle \to_{\mathcal{R}^r_{Orc}} \langle pub^{\tau}(f' > x > g \mid g\{c/x\}), tr : t \mid r\rangle \\
&\quad = \langle f' > x > g \mid g\{c/x\}, tr : t.\tau \mid r'\rangle
\end{aligned}$$

($\Longrightarrow$) This direction can be proved by simply reversing the argument given above.

(3), (5), and (6) are similar to (1) above, replacing parallel composition with sequential or asymmetric parallel composition and using the appropriate equations in $\mathcal{R}^r_{Orc}$, while (4) is similar to (2). $\square$

Proof of Lemma 4

Proof. ($\Longrightarrow$) By induction on a proof of $\mathcal{R}^s_{Orc} \vdash eagerEnabled(\mathcal{C}) = true$. There are four base cases:

1. $\mathcal{C} = \langle M(C), r\rangle$ (the [SiteCall] case),
2. $\mathcal{C} = \langle ?h, r\rangle$ (the [SiteRet] case),
3. $\mathcal{C} = \langle !c, r\rangle$ (the [Pub] case)
4. $\mathcal{C} = \langle E(P), r\rangle$ (the [Def] case)

The first three cases follow trivially from the equations defining active expressions and configurations in $\mathcal{R}^r_{Orc}$ and the fact that $\mathcal{C}$ is well-formed. The fourth case also follows trivially from the equations defining active expressions and configurations in $\mathcal{R}^r_{Orc}$ and the assumption that $\mathcal{C}$ is closed.

Now suppose $\mathcal{C} = \langle f \mid g, tr : t \mid r\rangle$ for non-zero expressions f and g. Then, $\mathcal{R}^s_{Orc} \vdash \mathit{eagerEnabled}(\langle f, tr : t \mid r\rangle) = \mathit{true}$, which by induction implies $\mathcal{R}^r_{Orc} \vdash \mathit{eager}(\langle f, tr : t \mid r\rangle) = \mathit{true}$, which by the equations of $\mathcal{R}^r_{Orc}$ implies $\mathcal{R}^r_{Orc} \vdash \mathit{eager}(\langle f \mid g, tr : t \mid r\rangle) = \mathit{true}$. The other two inductive cases for sequential and asymmetric parallel composition are similar.

($\Longleftarrow$) This direction can be proved by case analysis on $\mathcal{C}$. There are two cases:

- Suppose $\mathcal{C} = \langle f, \mathit{msg} : (\rho\ [\mathit{self}, c, h]) \mid r\rangle$ with h referenced in f is true. Then we proceed by induction on f. If $f = ?h$, then by the well-formedness of $\mathcal{C}$ it follows immediately that $\mathit{eagerEnabled}(\mathcal{C}) =$ true is provable from $\mathcal{R}^s_{Orc}$ (Note that this is the only possible base case as $?h$ is the only base expression that satisfies the condition h in f). Now suppose $f = f_1 \mid f_2$. Then, (modulo commutativity) h in f_1 holds and thus $\langle f_1, \mathit{msg} : (\rho\ [\mathit{self}, c, h]) \mid r\rangle$ is an eager configuration in $\mathcal{R}^r_{Orc}$, which by induction implies $\mathcal{R}^s_{Orc} \vdash \mathit{eagerEnabled}(\langle f, \mathit{msg} : (\rho\ [\mathit{self}, c, h]) \mid r\rangle) =$ true and hence the desired result. The other two inductive cases are similar.

- Suppose $\mathcal{C} = \langle f, r\rangle$ with f an active expression. Then we use induction on f again. By definition, the base expressions that are active are $!c$, $M(C)$ and $E(P)$. The assumption of $\mathcal{C}$ being closed implies $\mathcal{R}^s_{Orc} \vdash$ *eagerEnabled*$(\langle E(P), r\rangle) = \mathit{true}$, while well-formedness is enough to imply that $\mathit{eagerEnabled}(\langle M(C), r\rangle) = \mathit{true}$ and $\mathit{eagerEnabled}(\langle !c, r\rangle) = \mathit{true}$ are provable from $\mathcal{R}^s_{Orc}$.

  Now suppose $f = f_1 \mid f_2$. Then, f_1 is active (modulo commutativity) and thus by induction $\mathit{eagerEnabled}(\langle f_1, r\rangle) =$ true is provable from $\mathcal{R}^s_{Orc}$, which implies the desired result. The other two inductive cases are similar.

□

Proof of Lemma 5

Proof. The proof is similar to that of Lemma 4. The only-if direction can be shown by induction on a proof of $\mathcal{R}^s_{Orc} \vdash \mathit{intAction}(\mathcal{C}) = \mathit{true}$, while the if direction is provable by case analysis on $\mathcal{C}$.

□

Proof of Theorem 1

Proof. ($\Longrightarrow$) By induction on a proof of $\mathcal{C} \rightarrow_{\mathcal{R}^s_{Orc}} \mathcal{C}'$. There are five base cases as follows.

1. [SiteCall]. If

$$\begin{aligned}&\{M(C), tr : t \mid msg : \rho \mid hdl : h_n \mid clk : c_m \mid r\}\\ &\qquad \to [?h_n, tr : t\ .\ M\langle C, h_n|m\rangle \mid msg : \rho\ [M, C, h_n] \mid\\ &\qquad\qquad hdl : h_{n+1} \mid clk : c_m \mid r]\end{aligned}$$

then,

$$\begin{aligned}&\langle M(C), tr : t \mid msg : \rho \mid hdl : h_n \mid clk : c_m \mid r\rangle \to_{\mathcal{R}^r_{Orc}}\\ &\qquad \langle sc^{\uparrow}(\gamma, M, C), tr : t \mid msg : \rho \mid hdl : h_n \mid clk : c_m \mid r\rangle\\ &\quad = \langle sc^{\downarrow}(\gamma, h_n), tr : (t.M\langle C, h_n \mid m\rangle) \mid msg : (\rho\ [M, C, h_n]) \mid\\ &\qquad hdl : h_{n+1} \mid ...\rangle\\ &\quad = \langle ?h_n, tr : (t.M\langle C, h_n \mid m\rangle) \mid msg : (\rho\ [M, C, h_n]) \mid hdl : h_{n+1} \mid ...\rangle\end{aligned}$$

2. [SiteRet]. Suppose

$$\begin{aligned}&\langle ?h, tr : t \mid msg : (\rho\ [self, c, h]) \mid clk : c_m \mid r\rangle \to_{\mathcal{R}^s_{Orc}}\\ &\qquad\qquad \langle !c, tr : (t.h?c|m) \mid msg : \rho \mid clk : c_m \mid r\rangle\end{aligned}$$

Then, $\mathcal{R}^s_{Orc}$ proves that the *intAction* predicate is not true for this configuration. By Lemma 5, this configuration is not active in $\mathcal{R}^r_{Orc}$, and since h in $?h = true$ is provable from $\mathcal{R}^r_{Orc}$, we get

$$\begin{aligned}&\langle ?h, tr : t \mid msg : (\rho\ [self, c, h]) \mid clk : c_m \mid r\rangle \to_{\mathcal{R}^r_{Orc}}\\ &\qquad \langle sr(?h, c, h), tr : (t.h?c|m) \mid msg : \rho \mid clk : c_m \mid r\rangle\\ &\quad = \langle !c, tr : (t.h?c|m) \mid msg : \rho \mid clk : c_m \mid r\rangle\end{aligned}$$

3. [Pub]. If

$$\langle !c, tr : t \mid clk : c_m \mid r\rangle \to_{\mathcal{R}^s_{Orc}} \langle \mathbf{0}, tr : t.(!c|m) \mid clk : c_m \mid r\rangle$$

then,

$$\begin{aligned}\langle !c, tr : t \mid clk : c_m \mid r\rangle &\to_{\mathcal{R}^r_{Orc}} \langle pub(\mathbf{0}, c), tr : t \mid clk : c_m \mid r\rangle\\ &= \langle \mathbf{0}, tr : t.(!c|m) \mid clk : c_m \mid r\rangle\end{aligned}$$

4. [Def]. Suppose

$$\begin{aligned}&\langle E(P), tr : t \mid con : (\sigma, E(Q) =_{def} f) \mid r\rangle \to_{\mathcal{R}^s_{Orc}}\\ &\qquad\qquad \langle f\{P/Q\}, tr : t.\tau \mid con : (\sigma, E(Q) =_{def} f) \mid r\rangle\end{aligned}$$

Then, we have

$$\begin{aligned}&\langle E(P), \mathit{tr}: t \mid \mathit{con}: (\sigma, E(Q) =_{def} f) \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}}\\ &\qquad\langle ec^{\uparrow}(\gamma, E, P), \mathit{tr}: t \mid \mathit{con}: (\sigma, E(Q) =_{def} f) \mid r\rangle\\ &\quad= \langle ec^{\downarrow}(\gamma, f\{P/Q\}), \mathit{tr}: t.\tau \mid \mathit{con}: (\sigma, E(Q) =_{def} f) \mid r\rangle\\ &\quad= \langle f\{P/Q\}, \mathit{tr}: t.\tau \mid \mathit{con}: (\sigma, E(Q) =_{def} f) \mid r\rangle\end{aligned}$$

5. [Tick]. Suppose

$$\langle f, \mathit{clk}: c_m \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f, \mathit{clk}: c_{m+1} \mid \delta(r)\rangle$$

Then, $\mathcal{R}^s_{Orc}$ proves $\mathit{eagerEnabled}(\langle f, (\mathit{clk}: c_m) \mid r\rangle) \neq \mathit{true}$. By lemma 4. this is equivalent to $\langle f, (\mathit{clk}: c_m) \mid r\rangle$ not being an eager configuration in $\mathcal{R}^r_{Orc}$. Therefore, we have

$$\langle f, \mathit{clk}: c_m \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f, \mathit{clk}: c_m \mid \delta(r)\rangle$$

For the inductive step, there are six cases:

1. [Sym]. Suppose f and g are non-zero expressions. Then,

$$\langle f \mid g, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f' \mid g, \mathit{tr}: t.L \mid r'\rangle$$

implies $\langle f, \mathit{tr}: \mathit{nil} \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f', (\mathit{tr}: L) \mid r'\rangle$. By lemma 2 and the inductive assumption, this implies $\langle f, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f', \mathit{tr}: t.L \mid r'\rangle$, which, by part (1) of lemma 3, holds if and only if

$$\langle f \mid g, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f' \mid g, \mathit{tr}: t.L \mid r'\rangle$$

We note that if L is a site return event, then $\mathit{active}(g) \neq \mathit{true}$, by assumption.

2. [Seq1V]. Suppose

$$\langle f > x > g, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f' > x > g \mid g\{c/x\}, \mathit{tr}: t.\tau \mid r'\rangle$$

Then, $\langle f, \mathit{tr}: \mathit{nil} \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f', \mathit{tr}: (!c|m) \mid r'\rangle$. By lemma 2 and the inductive assumption, this implies $\langle f, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f', \mathit{tr}: t.(!c|m) \mid r'\rangle$, which, by part (2) of lemma 3, holds if and only if

$$\langle f > x > g, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f' > x > g \mid g\{c/x\}, \mathit{tr}: t.\tau \mid r'\rangle$$

3. [Seq1N]. Suppose L is not a publishing event. Then,

$$\langle f > x > g, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f' > x > g, \mathit{tr}: t.L \mid r'\rangle$$

implies $\langle f, \mathit{tr}: \mathit{nil} \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f', (\mathit{tr}: L) \mid r'\rangle$. By lemma 2 and the inductive assumption, this implies $\langle f, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f', \mathit{tr}: t.L \mid r'\rangle$, which, by part (3) of lemma 3, holds if and only if

$$\langle f > x > g, \mathit{tr}: t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f' > x > g, \mathit{tr}: t.L \mid r'\rangle$$

4. [Asym1V]. Suppose

$$\langle g \text{ where } x :\in f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle g\{c/x\}, tr : t.\tau \mid r'\rangle$$

Then, $\langle f, tr : nil \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f', tr : (!c|m) \mid r'\rangle$. By lemma 2 and the inductive assumption, this implies $\langle f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f', tr : t.(!c|m) \mid r'\rangle$, which, by part (4) of lemma 3, holds if and only if

$$\langle g \text{ where } x :\in f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle g\{c/x\}, tr : t.\tau \mid r'\rangle$$

5. [Asym1N]. Suppose L is not a publishing event. Then,

$$\langle g \text{ where } x :\in f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle g \text{ where } x :\in f', tr : t.L \mid r'\rangle$$

implies $\langle f, tr : nil \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f', (tr : L \mid r'\rangle$. By lemma 2 and the inductive assumption, this implies $\langle f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f', tr : t.L \mid r'\rangle$, which, by part (5) of lemma 3, holds if and only if

$$\langle g \text{ where } x :\in f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle g \text{ where } x :\in f', tr : t.L \mid r'\rangle$$

If L is a site return event, then $active(g) \neq true$, by assumption.

6. [Asym2]. Suppose

$$\langle g \text{ where } x :\in f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle g' \text{ where } x :\in f, tr : t.L \mid r'\rangle$$

Then, $\langle g, tr : nil \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle g', (tr : L \mid r'\rangle$. By lemma 2 and the inductive assumption, this implies $\langle g, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle g', tr : t.L \mid r'\rangle$, which, by part (6) of lemma 3, holds if and only if

$$\langle g \text{ where } x :\in f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle g' \text{ where } x :\in f, tr : t.L \mid r'\rangle$$

Again, if L is a site return event, then $active(g) \neq true$, by assumption.

($\Longleftarrow$) If $\mathcal{C} \rightarrow_{\mathcal{R}^r_{Orc}} \mathcal{C}'$ is an instance of the [Tick] rule ($\mathcal{C}$ is of the form $\langle f, clk : c_m \mid r\rangle$ with it not being an eager configuration), then the implication holds trivially by the corresponding [Tick] rule in $\mathcal{R}^s_{Orc}$. So, suppose the hypothesis is not an instance of the tick rule. Then, we observe that the hypothesis must be of the form $\langle f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f, tr : t.L \mid r\rangle$, i.e., $\mathcal{C}$ evolves to $\mathcal{C}'$ while generating an event. To complete the proof, we proceed by induction on f.

If f is a base expression (a site call, a site return, a publishing expression, or an expression call), then the implication holds easily by the equations in $\mathcal{R}^r_{Orc}$ and the assumption that $\mathcal{C}$ is closed.

Suppose the hypothesis is of the form $\langle f \mid g, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f' \mid g, tr : t.L \mid r'\rangle$. Then by part (1) of lemma 3, this implies $\langle f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f', tr : t.L \mid r'\rangle$, which by the induction hypothesis implies $\langle f, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f', tr : t.L \mid r'\rangle$, and thus the conclusion holds (when L is a site return event, $active(g) \neq true$, by assumption).

Suppose the expression component of $\mathcal{C}$ is of the form $f > x > g$. If L is a publishing event, then the hypothesis is of the form

$$\langle f > x > g, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f' > x > g \mid g\{c/x\}, tr : t.\tau \mid r'\rangle$$

which by Lemma 3, part 2, is equivalent to $\langle f, tr : nil \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f', tr : (!c|m) \mid r'\rangle$, implying that $\langle f, tr : nil \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f', tr : (!c|m) \mid r'\rangle$, and thus $\langle f > x > g, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^s_{Orc}} \langle f' > x > g \mid g\{c/x\}, tr : t.\tau \mid r'\rangle$. Otherwise, if L is not a publishing event, then the hypothesis is of the form

$$\langle f > x > g, tr : t \mid r\rangle \rightarrow_{\mathcal{R}^r_{Orc}} \langle f' > x > g, tr : t.L \mid r'\rangle$$

which, by part 3 of Lemma 3 and the inductive assumption, implies the desired conclusion.

The third case for the asymmetric parallel composition is similar and follows by parts 4, 5 and 6 of lemma 3 and induction. $\square$