Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis

Musab AlTurki and José Meseguer

The University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
{alturki,meseguer}@cs.illinois.edu

April 19, 2010

Abstract. Orc is a theory of orchestration of services that allows structured programming of distributed and timed computations. Several formal semantics have been proposed for Orc, including a rewriting logic semantics developed by the authors. Orc also has a fully fledged implementation in Java with functional programming features. However, as with descriptions of most distributed languages, there exists a fairly substantial gap between Orc's formal semantics and its implementation, in that: (i) programs in Orc are not easily deployable in a distributed implementation just by using Orc's formal semantics, and (ii) they are not readily formally analyzable at the level of a distributed Orc implementation. In this work, we overcome problems (i) and (ii) for Orc. Specifically, we describe an implementation technique based on rewriting logic and Maude that narrows this gap considerably. The enabling feature of this technique is Maude's support for external objects through TCP sockets. We describe how sockets are used to implement Orc site calls and returns, and to provide real-time timing information to Orc expressions and sites. We then show how Orc programs in the resulting distributed implementation can be formally analyzed at a reasonable level of abstraction, and discuss the assumptions under which the analysis can be deemed correct. Finally, the distributed implementation and the formal analysis methodology are illustrated with a case study.

1 Introduction

Concurrent languages for new application areas such as web services pose interesting challenges. The usefulness of such languages is quite clear, but their correctness, both that of a language implementation and of specific programs in the language, is crucial for their safety and security. In particular, one would like to have, among other things: (i) a clear, semantics-preserving path from a language specification to a distributed language implementation; (ii) furthermore, this distributed implementation path should come with *formal correctness guarantees*; and (iii) it should be possible to *formally verify* that specific programs written in such a language, and implemented according to the above path from a language specification, satisfy appropriate formal requirements. However, the distributed nature of such languages makes tasks (i)–(iii) more challenging than for sequential languages.

This report addresses all these issues for the Orc programming language [1, 2], an elegant and powerful language for orchestration of web services. Orc has a well-developed theoretical basis [2–6] and also a well-engineered language implementation [7, 8]. But as for any other language, there is a substantial gap between a language's theoretical description and its implementation. Ideally, we would like to reason, and obtain formal guarantees about, Orc programs in their implemented form, but this is not a trivial matter.

Our Approach. In this report we propose and demonstrate the effectiveness of a method to substantially narrow the gap between the theoretical level of a distributed language like Orc and an actual implementation. Our method is *semantics-based* and builds on our earlier work on a *rewriting-logic semantics* of Orc [9], in which we showed how subtle issues about the Orc semantics, including its real-time nature, and the essential priority that internal events in an Orc expression should have over external communication, could be faithfully modeled in our real-time rewriting semantics. It also builds on our subsequent work giving to Orc a more efficient *reduction semantics* [10, 11], which brought the Orc language definition substantially closer to an actual distributed implementation while preserving semantic equivalence. The third step, taken in this work, is to pass from a *rewriting-based reduction semantics* to a *rewriting-based implementation* of such a semantics in a seamless way. Since the original SOS-like semantics of Orc and the reduction semantics are semantically equivalent [11], this substantially narrows the gap between the language's formal semantics and its implementation.

How is this correct-by-construction implementation accomplished? The key idea is that concurrent rewriting is *both* a theoretical model and a practical method of distributed computation. Specifically, in the Maude language [12], a high-performance implementation of rewriting logic, asynchronous message-passing between distributed objects is accomplished by concurrent rewriting via sockets. For Orc, the objects are either Orc expressions, which play the role of clients, or web sites, which play the role of servers. But since for Orc real time is of the essence, an important issue that must be addressed is how real time is supported in the implementation. Here, the key observation is that Orc programs assume an asynchronous and possibly unreliable distributed environment such as the Internet, and therefore implicitly rely on their *local* notion of time for their computations. As a consequence, time is supported by *local ticker objects*, that interact in a tightly-coupled way with their co-located Orc objects.

How about formal verification of Orc programs? Specifically, how can we model check the formal requirements of an Orc program running in the rewriting-based distributed implementation just described? The answer is that we cannot model check such a program *directly* with existing tools, but that we can however model check it *indirectly*. The idea is in a sense quite simple, namely, we can *formally specify* both the internet sockets supporting the actual distributed implementation, and the ticker objects supporting the real-time behavior of Orc expressions. In this way, both distributed message-passing computation between Orc expression clients and web sites, and time elapse are faithfully *simulated* in the formal specification, in which our desired program can then be model checked. As we explain in the report, under reasonable assumptions about the granularity of time chosen for the tickers and the code of the Orc expressions, this simulated formal analysis gives us corresponding guarantees about the actual Orc programs running in the actual distributed Orc implementation.

Our Contributions. Our *first*, most obvious contribution is the *correct-by-construction* nature of our distributed Orc implementation, henceforth referred to as Dist-Orc. As explained in Section 3, this contribution builds on our previous results on three real-time rewriting semantics for Orc: (i) an SOS semantics; (ii) a reduction semantics; and (iii) an object-based semantics (which provides a substantial *extension* of Orc to explicitly model communication with *sites*). It also builds on the semantic equivalences already proved in [10] between the semantic levels (i)–(iii). In this way, a seamless path from formal language definition to a correct distributed implementation is obtained.

This first contribution would have remained at the theoretical level without a *second*, important contribution: this work shows for the first time how a rewriting logic specification of an object-oriented real-time system can be naturally transformed into an actual *distributed implementation* of such a system in physical time. For untimed object-based systems it was known that Maude sockets could be used for this purpose; but no technique was known for seamlessly passing from real-time specifications to their implementations. This is shown here for Dist-Orc, but the method is much more general and has already been applied to other real-time systems, like medical devices, in recent work by Sun and Meseguer [13].

A *third* important contribution is to show that we can still formally verify correctness properties of a distributed real-time *implementation* by modeling the implementation itself at an appropriately chosen level of abstraction. This had been done for untimed systems such as Mobile Maude [12], but it is done here for real-time systems for the first time.

The report is organized as follows. In Section 2, we give an overview of Orc and its semantics. Section 3 reviews the rewriting logic semantics of Orc previously developed and briefly discusses its specification in Maude. In Section 4, we present the distributed implementation Dist-Orc, explain in some detail its specification in Maude, and introduce the Dist-Auction example. Section 5 describes a formal model of Dist-Orc in Real-Time Maude and uses it to verify properties about Dist-Auction. The report concludes with a summary and a discussion of future work.

2 An Overview of Orc

Orc is a theory of orchestration of services that provides an elegant model for describing concurrent computations. Orc uses the notion of a *site* to represent a general service, which may vastly range in complexity from a simple function to a complex web search engine, depending on the orchestration problem at hand. A site may also be used to represent the interaction with a human being, most commonly within the context of business workflows [14]. A site, when called, produces at most one value. A site may not respond to a call, either by design or as a result of a communication problem. For example, if *CNN* is a site that returns the news page for a given date d, then $CNN(d)$ might not respond because of a network failure or it may choose to remain silent because of an invalid input value d. When a site responds to a call with some value c, it is said to *publish* the value c. Site calls are *strict*, in that a site call cannot be initiated before its parameters are bound to concrete values, e.g., the call $CNN(x)$, with x a variable, remains blocked until x is bound to some concrete value.

Orc's computation model is timed. Different site calls may occur at different times. A site call may be purposefully delayed using the *internal* site $rtimer(t)$, which publishes a signal after t time units. Furthermore, responses from calls to *external* sites may experience unpredictable delays and communication failures, which could affect whether and when other site calls are made. Unlike external sites, however, responses from internal sites, such as *rtimer*, are assumed to have completely predictable timed behaviors; for example, $rtimer(t)$ will publish a signal in exactly t time units. Orc also assumes a few more internal sites, which are needed for effective programming in Orc. They are: (1) the $if(b)$ site, which publishes a signal if b is true and remains silent otherwise, (2) $let(\bar{x})$, which publishes a tuple of the list of values in $\bar{x}$, or the value of $\bar{x}$ itself if $|\bar{x}| = 1$, (3) *Signal*, which publishes a signal immediately (equivalent to $if(true)$), and (4) *Clock*, which publishes the current time value.

2.1 The Orc Language

Orchestration *expressions* in Orc describe how individual site calls (and responses) are combined in order to accomplish a larger, more useful task. Orc expressions are built up from site calls using three combinators: sequential composition $> x >$, symmetric parallel composition $|$, and asymmetric parallel composition $< x <$. As shown in [2], with these three combinators, Orc is able to express a wide variety of distributed computations succinctly and elegantly.

$$\begin{array}{rl}
\text{Orc program} & ::= \bar{d}\ ;\ f \\
d \in \text{Declaration} & ::= E(\bar{x}) =_{def} f \\
f, g \in \text{Expression} & ::= \mathbf{0} \mid M(\bar{p}) \mid E(\bar{p}) \mid \quad f \mid g \quad \mid \quad f > x > g \quad \mid \quad g < x < f \\
p \in \text{Actual Parameter} & ::= x \mid c \mid M
\end{array}$$

Fig. 1. Syntax of Orc

The syntax of Orc is shown in Figure 1. An Orc *program* consists of an optional list of declarations, giving names to expressions, followed by an Orc expression to be evaluated. Declarations promote modular specifications and allow (mutually) recursive expressions. An *expression* can be either: (1) the silent expression ($\mathbf{0}$), which represents a site that never responds; (2) a site or an expression call having an optional list of actual parameters; or (3) the composition of two expressions by one of the three composition operators:

Symmetric parallel composition, $f \mid g$, which models concurrent execution of independent threads of computation. For example, $\mathit{CNN}(d) \mid \mathit{BBC}(d)$, where *CNN* and *BBC* are sites, calls both sites concurrently and may publish up to two values (news pages in this example) depending on the publication behavior of the individual sites.

Sequential composition, $f > x > g$, which executes f, and for every value c published by f, a fresh instance of g, with x bound to c, is created and run in parallel with $f > x > g$. This generalizes sequencing expressions in traditional programming languages, where f publishes exactly one value. For example, if $\mathit{Email}(a,x)$ is a site that sends an e-mail message given by x to a fixed address a, then the expression $\mathit{CNN}(d) > x > \mathit{Email}(a,x)$ may cause a news page to be sent to a. If $\mathit{CNN}(d)$ does not publish a value, $\mathit{Email}(a,x)$ is never invoked. Similarly, the expression $(\mathit{CNN}(d) \mid \mathit{BBC}(d)) > x > \mathit{Email}(a,x)$ may result in sending zero, one, or two messages to a.

Asymmetric parallel composition, $f < x < g$, which executes f and g concurrently but terminates g once it has published its first value, which is then bound to x in f. For instance, the expression

$$\mathit{Email}(a,x) < x < (\mathit{CNN}(d) \mid \mathit{BBC}(d))$$

sends at most one e-mail message to a, depending on which site publishes a value first. If neither site publishes a value, the variable x is not bound to a concrete value and, therefore, the call to *Email* is never made.

As can be noted from the informal description above, the execution of an Orc expression may in general involve several concurrently running threads, and may result in publishing a (time-ordered) stream of values. Moreover, a variable x may only occur *bound* in an expression g by either the sequential composition $f > x > g$ or the asymmetric parallel composition $g < x < f$. If a variable is not bound, it is said to be *free*.

To minimize use of parentheses, we assume that sequential composition has precedence over symmetric parallel composition, which has precedence over asymmetric composition. We also use the syntactic sugar $f \gg g$ for sequential composition when no value passing from f to g is taking place, which corresponds to the case of a sequential composition $f > x > g$ where x is *not a free* variable of g.

To illustrate these ideas, we list a few small examples that can be found, among many other examples, in [1, 2]. First, the Orc expression below specifies a timeout on the call to a site M.

$$\mathit{let}(x) < x < (\, M \mid \mathit{rtimer}(t) \gg \mathit{let}(0)\,)$$

Upon executing the expression, both sites M and *rtimer* are called. If M publishes a value c before t time units, then c is the value published by the expression.

But if M publishes c in exactly t time unites, then either c or 0 is published. Otherwise, 0 is published.

Common programming idioms can also be easily and succinctly specified using Orc's combinators. For instance, the two-branch conditional statement **if** b **then** f **else** g, with b a boolean value, can be written in Orc as the following expression

$$\mathit{if}(b) \gg f \;\mid\; \mathit{if}(\neg b) \gg g$$

Given the behavior of the internal site *if*, only one of f and g is executed, depending on the truth value of b.

Another example is the following Orc expression declaration, which defines an expression that recursively publishes a signal every t time units, indefinitely.

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

The expression named *Metronome* can be used to repeatedly initiate an instance of a task every t time units. For example, the expression $\mathit{Metronome}(100) \gg \mathit{UpdateLocation}$ calls on the task of updating the current location of a mobile user every hundred time units.

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

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

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

$$\frac{f \overset{l}{\hookrightarrow} f'}{f \mid g \overset{l}{\hookrightarrow} f' \mid g}\ (\textsc{Sym1}) \qquad \frac{f \overset{l}{\hookrightarrow} f' \qquad l \neq !c}{g < x < f \overset{l}{\hookrightarrow} g < x < f'}\ (\textsc{Asym1N})$$

$$\frac{g \overset{l}{\hookrightarrow} g'}{f \mid g \overset{l}{\hookrightarrow} f \mid g'}\ (\textsc{Sym2}) \qquad \frac{g \overset{l}{\hookrightarrow} g'}{g < x < f \overset{l}{\hookrightarrow} g' < x < f}\ (\textsc{Asym2})$$

Fig. 2. Asynchronous semantics of Orc

A formal description of the (untimed) asynchronous semantics of Orc was given as an SOS specification in [2], and is shown here in Figure 2. The semantics was given as an SOS specification defining a labeled transition system with four labels: (1) a site call label $M\langle c, u\rangle$, with u a fresh handle name uniquely identifying the call that caused it, (2) a site return label $u?c$, (3) a label $!c$ for publishing a value, and (4) the internal (unobservable) transition label τ. The reader is referred to [2], for a detailed discussion of the specification. Here, only a few subtleties are emphasized. We first note that symmetric parallel composition

$f \mid g$ in Orc is similar to that of a process calculus in which both expressions f and g can evolve concurrently without any restriction. However, in asymmetric parallel composition $g < x < f$, once f publishes its first value (the ASYM1V rule), the remaining computations of f are discarded and the published value is bound to x in g. Therefore, it is possible for some computations of g to be blocked waiting for a value for x. We also note that in sequential composition $f > x > g$, a new instance of g is created for every value published by f (the SEQ1V rule), which generalizes the usual notion of sequential composition in sequential programming languages.

2.2 An Auction Management Example

This section concludes with a description of a larger example in Orc, namely AUCTION, which will be the basis for a case study illustrating DIST-ORC in Section 4, and its formal analysis in Section 5. The Orc program AUCTION, which was originally inspired by the auction example in [2], is a simplified online auction management application that manages posting new items for auction, coordinates the bidding process, and announces winners.

AUCTION assumes the following sites: (1) A *Seller* site, which maintains a list of items to be auctioned and responds to the message postNext by publishing the next available item, (2) a *Bidders* site, which maintains a list of bidders and their bids, and responds to the message nextBidList, which solicits a list of higher bids for the auctioned item, (3) a *MaxBid* site, which is a functional site that publishes the highest bid of a list of bids, and (4) an *Auction* site, which maintains a list of available items and responds to post and getNext for adding and retrieving an item from the list, respectively. The *Auction* site also keeps a list of winners and the respective items won, and responds to the message won, which declares a bidder as the winner for the auctioned item.

$$
\begin{aligned}
&\mathit{Posting}(\mathit{seller}) =_{def} \mathit{seller}(\text{“}\mathsf{postNext}\text{”}) > x > \mathit{Auction}(\text{“}\mathsf{post}\text{”}, x) \gg \mathit{rtimer}(1) \gg \\
&\qquad \mathit{Posting}(\mathit{seller})\\
&\mathit{Bidding} =_{def} \mathit{Auction}(\text{“}\mathsf{getNext}\text{”}) > (\mathit{id}, d, m) > \mathit{Bids}(\mathit{id}, d, m, 0) > (\mathit{wn}, \mathit{wb}) > \\
&\qquad (\ \mathit{if}(\mathit{wn} = 0) \gg \mathit{Bidding}()\\
&\qquad \mid \mathit{if}(\mathit{wn} \neq 0) \gg \mathit{Auction}(\text{“}\mathsf{won}\text{”}, \mathit{wn}, \mathit{id}, \mathit{wb}) \gg \mathit{Bidding}()\ \ )\\
&\mathit{Bids}(\mathit{id}, d, \mathit{wb}, \mathit{wn}) =_{def} (\ \mathit{if}(d \leq 0) \gg \mathit{let}(\mathit{wb}, \mathit{wn})\\
&\quad \mid \mathit{if}(d > 0) \gg \mathit{clock}() > t_a > \mathit{min}(d, 1) > t > \mathit{TimeoutRound}(\mathit{id}, \mathit{wb}, t) > x > \\
&\qquad (\ \mathit{if}(x = \mathit{signal}) \gg \mathit{Bids}(\mathit{id}, d - t, \mathit{wb}, \mathit{wn})\\
&\qquad \mid \mathit{if}(x \neq \mathit{signal}) \gg \mathit{rtimer}(1) \gg \mathit{clock}() > t_b > \mathit{Bids}(\mathit{id}, d - (t_b - t_a), x_0, x_1)))\\
&\mathit{TimeoutRound}(\mathit{id}, \mathit{bid}, t) =_{def}\\
&\qquad \mathit{let}(x) < x < (\ \mathit{rtimer}(t)\ \mid\ \mathit{Bidders}(\text{“}\mathsf{nextBidList}\text{”}, \mathit{id}, \mathit{bid}) > \mathit{bl} > \mathit{MaxBid}(\mathit{bl})\ )
\end{aligned}
$$

Fig. 3. Orc expressions in the AUCTION program

Figure 3 lists Orc expressions used by AUCTION. The *Posting* expression recursively queries a given seller site for the next item available for auction x, and then posts it to the auction by calling the *Auction* site. The call to *Auction* blocks until bidding on x has ended. The *Posting* expression then waits for one more time unit before querying the seller for the next item.

The *Bidding* expression recursively queries the auction site for the next item available for auction, where an item is a tuple (id, d, m), with id the item identifier, d its auction duration, and m the starting bid. The expression then collects bids for the item in rounds from the bidders site for the duration of the auction, where each round lasts for a maximum of one unit of time. Once the bidding ends, the *Bidding* expression announces the winning bidder name wn, the item id and the winning bid wb before proceeding to the next item. In Figure 3, we use pattern matching syntax for tuple variables, and integer subscripts in variable names to pick elements of the underlying tuple, e.g. x_0 is the first element of the tuple given by x, and so on.

The declarations in Figure 3 along with the expression $Posting(s) \mid Bidding$, for a given seller site s, specify in Orc the *Auction* program.

3 Rewriting Semantics of Orc

Rewriting logic [15] is a general semantic framework that is well suited for giving formal definitions of programming languages and systems, including their concurrent and real-time features (see [16, 17] and references there, and [18]). Furthermore, with the availability of high-performance rewriting logic implementations, such as Maude [12], language specifications can both be executed and model checked.

In previous work [10], we have developed an executable specification giving a formal semantics to Orc in rewriting logic using Maude. The specification was shown to capture Orc's intended *synchronous* semantics [11], where actions internal to an Orc expression, namely site calls, expression calls, and publishing of values, are given priority over interactions with the environment (processing responses from external sites), while also capturing its timed behaviors. Furthermore, our specification went through three main semantics-preserving refinements in order to achieve greater efficiency and expressiveness by exploiting rewriting logic's distinction between *equations* and *rules*. Maude's formal analysis tools, including its LTL model checker, were then used to verify various properties about Orc programs.

3.1 Rewrite Theories and Orc's Rewriting Logic Semantics

The formal semantics of Orc is specified in rewriting logic as a rewrite theory $\mathcal{R}_{Orc} = (\Sigma_{Orc}, E_{Orc}, R_{Orc})$, where (i) Σ_{Orc} is a *signature* declaring the sorts and operators to be used in the language specification, (ii) E_{Orc} is a set of *equations* that algebraically specify the properties satisfied by these operators, and (iii) R_{Orc} is a set of *rewrite rules* defining the computational behavior of the language.

Thus, while the equational theory (Σ_{Orc}, E_{Orc}) specifies the static state structure of programs in Orc, the rules R_{Orc} specify their externally observable dynamic behavior.

In this section, we briefly review at a high level the rewriting logic semantics of Orc. More details can be found in [10, 9, 11].

The SOS-based Semantics $\mathcal{R}^{\mathbf{SOS}}_{\mathbf{Orc}}$ [9]. This specification is directly based on the asynchronous SOS semantics of Orc given in [2] and summarized in Figure 2. It was obtained using a general, semantics-preserving transformation method from modular structural operational semantics (MSOS) to rewriting logic [19]. The specification was also refined to capture Orc's intended synchronous and timed semantics. Being based on the SOS semantics, the specification was relatively easy to develop and understand, and its correctness followed almost entirely from the correctness of the transformation used. However, $\mathcal{R}^{SOS}_{Orc}$ made extensive use of top-level, conditional rewrite rules with rewrite conditions, corresponding to the rules in the SOS specifications, which in practice resulted in a fairly inefficient executable and analyzable semantics.

The Reduction Semantics $\mathcal{R}^{\mathbf{Red}}_{\mathbf{Orc}}$ [10]. This specification was developed to alleviate the efficiency problems with $\mathcal{R}^{SOS}_{Orc}$ while preserving its semantics. This was achieved mainly by exploiting the distinction between rules and equations in rewriting logic and its inherently distributed semantics. Indeed, rewrite rules in $\mathcal{R}^{Red}_{Orc}$ are much simpler and more localized. This resulted in a much more efficiently executable semantics in practice, which was demonstrated through different examples in [10]. Moreover, the specification was shown to be strongly bisimilar to $\mathcal{R}^{SOS}_{Orc}$, which implied that Orc's intended semantics was persevered [11].

The Object-based Semantics $\mathcal{R}_{\mathbf{Orc}}$. The object-based semantics, which was first introduced in [10], generalizes the reduction semantics to multiple Orc expressions and makes explicit the interactions between Orc site and expression objects. Unlike the reduction semantics, the environment of an Orc expression in $\mathcal{R}_{Orc}$ is no longer a black box, but is instead modeled explicitly by site objects and by modeling asynchronous message passing. The extension to object-based semantics enabled defining arbitrarily complex and distributed applications in Orc, in which the semantics of an individual Orc expression is given precisely by the previous reduction semantics $\mathcal{R}^{Red}_{Orc}$. The distributed implementation presented in this report builds on the object-based semantics given by $\mathcal{R}_{Orc}$. We, therefore, give a quick overview of the object-based semantics below.

In $\mathcal{R}_{Orc}$ [10], the state of an Orc program is defined as a *configuration* (of sort *Conf*) of objects and messages. Configurations are multisets specified by the associative and commutative empty juxtaposition operator $\_\_ : \mathit{Conf}\ \mathit{Conf} \to \mathit{Conf}$, with the empty multiset *null* as the identity element. An object is a term of the form $\langle I : C \mid AS \rangle$, with I a unique object identifier, C the class name of the object, and AS a set of attribute-value pairs, each of the form *attr* : *value*, where *attr* is the attribute's *name*, and *value* is the corresponding *value*. The sets AS are algebraically built up with the union operator $\_,\_ : \mathit{AttributeSet}\ \mathit{AttributeSet} \to \mathit{AttributeSet}$. In our object-based rewriting semantics, there are two main classes of Orc objects: *Expression* objects and *Site* objects, corresponding, respectively, to Orc expressions and sites. An Orc expression object maintains the Orc expression to be evaluated in an attribute *exp* as well as a set of expression declarations in the environment attribute *env*, while an Orc site object maintains the site's state in an abstract *state* attribute.

In keeping with the philosophy of the Orc theory, expression objects are modeled as active objects (or actors in the actor model) having a state and one or more threads of control (given as an Orc expression), and are capable of initiating (asynchronous) message exchange. Site objects, on the other hand, are reactive objects having internal states but are only capable of responding to incoming requests. They can be thought of as actors that have a passive-reactive behavior. In order to capture timing behaviors, an additional simple *Clock* object is also included in the configuration.

Messages in an Orc configuration are either site call or site return messages. Within an expression object I, a site call expression $M(C)$, with M the site object id and C a list of values, causes a site call message of the form $M \leftarrow sc(I, C)$ to be emitted into the configuration, which upon being received and processed by site M may in turn cause a site return message $I \leftarrow sr(M, c)$, with c the value published by M.

The semantics of Orc is then specified using both equations and rewrite rules. This combination of equations and rules is useful in deciding the level of abstraction at which the semantics is designed (see [17]). Of course, the choice in this case was guided by the previous structural operational semantics for Orc [2, 6], shown in Figure 2. In particular, the semantics captures four basic actions an Orc configuration may take: (1) a site call, (2) an expression call, (3) publishing of a value, and (4) a site return (which amounts to consuming a response from a site). As the *synchronous* semantics stipulates [2], the first three actions are *internal* to the expression object in which they take place, while site returns are considered *external* and have lower priority. Moreover, the semantics specifies the timed behavior of Orc using a *tick* rule that advances logical time and manages the effect of time elapse. To provide proper timing guarantees for internal sites and to rule out uninteresting behaviors, a rule application strategy that gives time ticking the lowest priority among all rules in the specification is used. For the analysis to be mechanizable, we also assume Orc programs with "non-Zeno" behaviors [20], such that only a finite number of internal (instantaneous) transitions are possible within any finite period of time[1].

In all three versions of the rewriting semantics of Orc outlined above, although many of the concepts and techniques related to specifying timed behaviors in rewrite theories were borrowed from work on *real-time rewrite theories* [18] and their implementations in Real-Time Maude [21], the tools and infrastructure provided by Core Maude were enough for our modeling and analysis purposes. However, in Section 5, we use Real-Time Maude to arrive at a more flexible and expressive formal model for the distributed implementation presented in the next section, and use its formal analysis tools, such as time-bounded model-checking.

[1] More details about these requirements were discussed in our earlier work [10].

3.2 Maude

Maude is a high-performance implementation of rewriting logic and its underlying membership equational sublogic, with syntax that is almost identical to the mathematical notation. The rewrite theory $\mathcal{R}_{Orc}$ specifying the semantics of Orc is given in Maude as a *system module*, declared with syntax `mod ORC is ... endm`, which may include submodules, sort, subsort, and operator declarations, equations, memberships and rewrite rules. To illustrate Maude's user-definable syntax, the fragment below specifies some of Orc's constants (note that Maude statements are terminated with a dot).

```
mod CONSTANTS is
  sorts Const ConstList .
  subsort Const < ConstList .
  op _,_ : ConstList ConstList -> ConstList [ctor assoc id: nilA] .
  op nilA : -> ConstList [ctor] .
  ...
  op signal :           -> Const [ctor] .
  op b      : Bool      -> Const [ctor] .
  op i      : Nat       -> Const [ctor] .
  op l      : ConstList -> Const [ctor] .
  op c      : String    -> Const [ctor] .
end
```

The module begins with a sort declaration for Orc constants `Const` and lists of constants `ConstList`, followed by a subsort declaration statement specifying that the sort of `Const` is a subsort of `ConstList`. As specified by the first operator declaration with the keyword `op`, a list of Orc constants (a term of the sort `ConstList`) is a comma-separated list of constants that is fully associative (given by the operator attribute `assoc`) and has the identity element `nilA` (given by the attribute `id:`). The `ctor` attribute declares a constructor operator (as opposed to a defined symbol). Some concrete Orc constants (elements of the sort `Const`) are then declared with appropriate constructors like `signal` and the operators `b`, `i`, `l`, `c`, which encapsulate the corresponding kinds of data such as boolean values, natural numbers, tuples, and encapsulated character strings.

Another example specification is a fragment defining part of the Orc expressions syntax and semantics, shown below.

```
mod EXPRESSION is
  protecting PARAMETER .
  sort Expr .
  op zero : -> Expr [ctor] .  ...
  op _>_>_ : Expr Var Expr -> Expr [ctor frozen (3)] .

  var x : Var . var f g : Expr .
  eq zero > x > f = zero .

  op _>>_ : Expr Expr -> Expr [frozen (2)] .
  eq f >> g = f > v('#!) > g .    --- assuming v('#) is never used in g
endm
```

The module inclusion statement `protecting PARAMETER` declares the current module `EXPRESSION` as extending the `PARAMETER` module, which defines Orc variables and actual parameters. This is followed by a sort declaration for Orc expressions `Expr`, followed by two operator declarations for the `zero` expression and sequential composition. The sequential composition operator is declared with the attribute `frozen(3)`, which declares rewrites forbidden on its third argument, since in sequential composition the expression on the right has no behavioral transitions. The module then specifies a semantic equation, declared with the keyword `eq`, asserting the left-identity property of sequential composition, with `x` and `f` as Maude meta-variables declared with the `var` keyword. Finally, a syntactic sugar for sequential composition with no value passing is introduced with the defined (non-constructor) operator `_>>_`. The Orc variable `v('#!)` is internal and is, therefore, assumed to never occur free in the right-hand side expression `g`.

Rewrite rules are introduced using the keyword `rl` (and `crl` for conditional rules). To illustrate our rewriting semantics specification in Maude, the following lists two rewrite rules capturing the action of a site call.

```
crl [SiteCall] :
 < O : Expr | exp: f , AS > => < O : Expr | exp: scallup(f', M, C) , AS >
    if f => scallup(f', M, C) .

 rl [SiteCall*] : M(C) => scallup(tmph, M, C) .
```

The first rule, labeled `SiteCall`, is a conditional rule that applies to any expression object whose Orc expression `f` given by its `exp` attribute is able to make a site call transition. The transition itself is characterized by the second rule `SiteCall*`, which transforms the site call to an auxiliary operator `scallup`, which is used to equationally specify how a site call and its effects are propagated across its context.

Rewriting logic specifications, such as $\mathcal{R}_{Orc}$, are executable in Maude. This provides the ability to simulate Orc programs by fair rewriting, using the command `frewrite`. Furthermore, specifications in Maude can be subjected to different formal analysis techniques. These include reachability analysis by breadth-first search with the `search` command, which can be used to verify violations of invariants, and linear temporal logic model checking for verifying more general properties, such as liveness and safety properties, using Maude's LTL model checker. For a complete description of Maude and its features, the reader is referred to [12], and for a description of Real-Time Maude and its formal analysis capabilities to [18].

4 A Distributed Implementation of Orc

The Orc theory was designed to specify, in a structured manner, concurrent computations, with emphasis on distribution through the notion of (external) sites. Furthermore, in practical applications it is typically the case that an Orc expression combines (through the use of Orc combinators) several subexpressions that independently orchestrate different but related tasks. For example, a typical online auction management expression may be composed of subexpressions managing: (1) seller inventories and product auction announcements, (2) bid collection and winner announcements, and (3) payments and shipping coordination. Such subexpressions need not be located on the same machine for the orchestration effort to be completed, but are, in fact, more often run on physically distributed autonomous agents spread across the web. We, therefore, describe an extension of the Orc theory to a distributed programming model that is both natural and useful in specifying and analyzing distributed computations with explicit treatment of external sites and messages. The extension encapsulates the Orc programming model as the underlying model for Orc expressions, and in this respect, its rewriting specification builds on the rewriting semantics specification of Orc $\mathcal{R}_{Orc}$ developed earlier [10], and briefly outlined in Section 3.

Indeed, the rewriting specification $\mathcal{R}_{Orc}$ paves the way for a rewriting-based, distributed implementation of Orc, Dist-Orc. As we show next, the transition from formal semantics to a distributed implementation is seamless and natural, thanks to Maude's internal support for external distributed objects through TCP sockets. The distributed implementation builds on the provably correct [11] rewriting semantics of Orc, $\mathcal{R}_{Orc}$, in Maude with only minor modifications, such as the changes of date representation needed to exchange data through sockets. This considerably increases our confidence in the correctness of the implementation and greatly narrows the gap between implementation and formal analysis.

In general, the method of transforming a real-time, object-based rewriting semantics into a real-time distributed implementation consists of three fundamental steps:

1. Defining the distributed structure of the system being specified by specifying locations and a globally unique naming mechanism for objects
2. Specifying the rewriting semantics of the communication model for distributed objects in the system
3. Devising a mechanism for capturing real, wall clock timing information and extending the rewriting semantics of time to incorporate this information

As explained below, a crucial enabling feature for steps (2) and (3) above is Maude's support for socket-based communication [12]. Through sockets, a Maude process is able to exchange messages with other processes, including other Maude instances, according to the connection-oriented TCP communication protocol.

We now discuss in some detail how this method is applied to Orc's rewriting semantics, outline the design and implementation choices in Dist-Orc and explain how they are specified in Maude. The full specification of Dist-Orc is given in Appendix A.

4.1 Distributed Orc Configurations

In the distributed implementation, an Orc configuration may span multiple nodes in an interconnected network, and is thus called a *distributed* Orc configuration. Both expression and external site objects in a distributed configuration are identified partly by their *location*, which is defined as a combination of an address (such as a URI or an IP address) and a port number.

```
sort Loc .     op l : String Nat -> Loc [ctor] .
```

To fully identify expression and external site objects, expression object identifiers, of sort `EOid`, and external site identifiers, of sort `XSOid`, also include a locally unique sequence number.

```
op S : Loc Nat -> XSOid [ctor] .     op E : Loc Nat -> EOid [ctor] .
```

Internal site objects, such as *if* and *rtimer*, are identified simply by their names, since their locations are implicit.

Within a distributed Orc configuration, a *localized configuration* (a term of sort `LocalSystem`) or simply a *configuration*, which is a configuration that is located at some node, is managed by an independent instance of Maude. In addition to expression and site objects, each such configuration contains a clock object for maintaining local time and a socket portal for exchanging messages with external objects in other configurations (more on this below).

```
sort LocalSystem .   op [_] : Configuration -> LocalSystem [ctor] .
```

The operator `[_]` encapsulates a localized configuration to support managing the local clock and the effects of time elapse (this is similar to the ideas presented in Real-Time Maude [18]).

4.2 Sockets and Messaging

A localized Orc configuration is declared to contain a *socket portal*, which is the predefined constant

```
<> : -> Portal [ctor] .
```

where `Portal` is a subsort of `Configuration`. The portal enables Orc objects to communicate with objects, such as sockets, which are external to the Maude configuration in which the portal resides.

In agreement with the Orc theory, the communication model between Orc expressions and sites follows very closely that of the client-server architecture, where Orc expressions are client objects requesting and processing services from sites as server objects. In particular, when an expression object `O` within some Orc configuration makes a site call to an external site `n` located at `l(SR, PT)`, with `SR` and `PT` the node's address and port, a site call message of the form `S(l(SR, PT), n) <- sc(O, C, h)` is created within the configuration, where `S(l(SR, PT), n)` is the site object identifier, as described above, and `h` is a temporary handle that uniquely identifies this call. This message then triggers the creation of a client socket to the called site through the following equation:

```
eq S(l(SR, PT), n) <- sc(OE, C, h, 0)
   = < P(OE,h) : Proxy | param : C, response : "" >
     createClientTcpSocket(socketManager, P(OE,h), SR, PT) .
```

Beside asking Maude's socket manager for a client socket to the site, the rule creates a temporary *proxy* object, which manages external communication for this particular site call on behalf of the expression object `O`. The proxy object also serves as a buffer for the site's response, since TCP sockets do not preserve message boundaries in general.

If a client TCP socket to `l(SR, PT)` is successfully created, Maude introduces the message `createdSocket(OP, socketManager, SC)` targeted to the proxy `OP` into the configuration, which causes the proxy to send the site call message to the external site object through the socket `SC`, as specified by the following rewrite rule:

```
rl [SendExtMessage] :
  createdSocket(OP, socketManager, SC) < OP : Proxy | param: C, AS >
  => < OP : Proxy | param: C, AS > send(SC, OP, (toString(C) + sep)) .
```

where `toString` is a function that properly serializes Orc constants into strings, so that they can be transmitted through sockets. The function uses a separator `sep` to distinguish message boundaries. At the other end, Orc constants are built back from such strings using another function `toConst`. Orc value serialization is described in detail in Section 4.3 below.

There is also the possibility of an unsuccessful client socket creation attempt due, for example, to an unavailable server or a network failure. In this case, Maude will report the error by issuing the message `socketError(OP, socketManager, S)`, with `S` a string describing the cause of the error. Such an error is a run-time error, which, for simplicity, is considered fatal in DIST-ORC, so that the site call and any subsequent transitions that depend on it will fail.

Once the site call message is sent, the reply `sent(OP, SC)` appears in the configuration and the proxy object waits for a response by introducing a `receive` message:

```
rl [RecExtMessage] :
  sent(OP, SC) < OP : Proxy | AS > => < OP : Proxy | AS > receive(SC, OP) .
```

When some string is received through the socket, the reply `received` appears, and the proxy object stores the received string in its buffer and waits for further input.

```
rl [RecExtMessageCont] :
  received(OP, OD, S) < OP : Proxy | param: C, response: S' >
    => < OP : Proxy | param: C, response: S' + S > receive(OD, OP)
```

The proxy will keep waiting for and accumulating input through the socket until it is remotely closed by the site, when the reply `closedSocket` appears. At this point, the site response is reconstructed and handed in to its expression object:

```
rl [ProcessExtMessage] :
  closedSocket(P(O,h), SC, S) < P(O,h) : Proxy | response: S' , AS >
    => OE <- sr(toConst(S'), h) .
```

On the server side, when a site is first initialized, it creates a server TCP socket, through which it keeps listening for incoming connections.

```
eq [InitializeSite] :
  < S(l(SR, PT), n) : Site | op : free, status : idle, AS >
  = < S(l(SR, PT), n) : Site | op : free, status : initializing, AS >
    createServerTcpSocket(socketManager, S(l(SR, PT), n), PT, 10) .

rl [CreatedServerSocket] :
  createdSocket(xOS, socketManager, LISTENER)
  < xOS : Site | op : free, status : initializing , AS >
  => < xOS : Site | op : free, status : active , AS >
     acceptClient(LISTENER, xOS) .
```

Once a client has been connected with a socket `CLIENT`, the site becomes ready for an incoming site call through `CLIENT`, while listening for other potential clients:

```
rl [AcceptedClient] :
  acceptedClient(xOS, LISTENER, IP, CLIENT)
  < xOS : Site | op : free, status : active, AS >
  => < xOS : Site | op : free, status : active , AS > receive(CLIENT, xOS)
     acceptClient(LISTENER, xOS) .
```

The site then accumulates the request from `CLIENT` (which contains the actual parameters for the site call):

```
crl [AccumulateRequest] :
  < xOS : Site | op : free , buffer : S, AS > received(xOS, CLIENT, S')
  => < xOS : Site | op : free , buffer : S + S', AS > receive(CLIENT, xOS)
        if find(S + S', sep, 0) == notFound .
```

The rule above buffers the serialized request by checking whether the message boundary indicator, given by `sep`, has been received. Once the message is received in its entirety, the following rule fires:

```
crl [PrepareReply] :
   received(xOS, CLIENT, S') < xOS : Site | op : free , buffer : S, AS >
   => < xOS : Site | op : exec(toConst(substr(S'', 0, length(S'')
                                    + (- length(sep)))), CLIENT),
           buffer : "", AS >
        if S'' := S + S' /\ n := find(S'', sep, 0) .
```

The rule causes the site to process the call using the function `exec(...)`, whose definition is site-dependent. When appropriate, the site might publish a value as a result of this site call, which then causes a response to be sent back to the client:

```
eq CLIENT <- sr(c, xOS, 0) < xOS : Site | AS >
  = < xOS : Site | AS > send(CLIENT, xOS, toString(c)) .
```

Once the response is sent, the site closes the socekt:

```
rl [ReplySent] :
  sent(xOS, CLIENT) < xOS : Site | AS >
  => < xOS : Site | AS > closeSocket(CLIENT, xOS) .

rl [ClosedClientSocket] :
  closedSocket(xOS, CLIENT, S) < xOS : Site | AS >
  => < xOS : Site | AS > .
```

Note that when a site remains silent, no response is generated, and the client will block waiting for a response through the other end of the open socket `CLIENT`.

It is worth noting here that, just like any other physically distributed communication mechanism, messaging through sockets is inherently prone to various potential communication problems. In addition to socket creation errors mentioned above, these include dropped connections, lossy channels and unpredictably long delays. In DIST-ORC, such problems are dynamic errors that might be exposed while executing a distributed Orc program, and typically cause the Orc objects in which they appear to fail.

4.3 Serialization of Orc Constants

Localized Orc configurations communicate with each other by external site calls and returns, which involve exchanging Orc constants either as actual parameters to calls or as return values. Since this communication takes place through Maude's TCP sockets, Orc constants need to be serialized as transferable strings just before they are transmitted, and then reconstructed back to Orc constants just after they are received. Similarly to how this issue was previously approached in Maude [22, 23], we utilize Maude's meta-level capabilities, provided by the module `META-LEVEL` to produce proper serialization and reconstruction procedures for Orc constants. More specifically, we define a module `CONST-STRING-CONVERSION` that imports both the `PARAMETER` module, containing declarations of Orc constant sorts and operators, and the `META-LEVEL` module.

```
mod CONST-STRING-CONVERSION is
  inc PARAMETER .
  pr META-LEVEL .
  ...
endm
```

Within this module, two main operators are defined: `toString` and `toConst`, which were briefly introduced above. The operator `toString` defines a partial function that attempts to convert an Orc constant or a list of constants `C` (a term of sort `ConstList`) into a string `S` in Maude.

```
op toString  : ConstList ~> String .
eq toString(c(S)) = S .
eq toString(C) = qidListString(metaPrettyPrint(
                 upModule('PARAMETER, false), upTerm(C), none)) [owise] .
```

Encapsulated strings of the form `c(S)` are easily converted into strings by using their underlying string values, while conversion of any other constant or list of constants is performed by building up a string out of their meta-representations. The operator `qidListString`, whose definition is given in Appendix A.3, creates a string out of the list of quoted identifiers returned by the meta-level operator `metaPrettyPrint` representing the different constants in `C` (see [12] for a detailed description of the meta-level operators).

The dual operator is `toConst`, which is defined in `CONST-STRING-CONVERSION` and is partially shown below.

```
 op toConst : String -> Const .
ceq toConst(S) = downTerm(getTerm(metaParse(
     upModule('PARAMETER, false), stringQidList(S), 'Const)), error(S))
     if substr(S, 0, 6) == "signal"   ---- signal constant
     or substr(S, 0, 4) == "b `("     ---- bool
     or substr(S, 0, 4) == "l `("     ---- tuple
     or substr(S, 0, 4) == "S `("     ---- site object id
     or substr(S, 0, 3) == "let"      ---- `let' site id
     ...
     or substr(S, 0, 7) == "_`,_ `(" .  ---- ConstList
 eq toConst(S) = c(S) [owise] .       ---- arbitrary string
```

If the input string `S` to `toConst(S)` matches the meta-representation of a known Orc constant (or list of constants), which is decided by examining an appropriate prefix of `S`, the first equation applies and the conversion process is carried out by Maude's meta-level operators. Otherwise, the string is considered an encapsulated Orc string constant. The operator `stringQidList` converts a string built by `qidListString` back into a list of quoted identifiers to be processed by the meta-level (see Appendix A.3).

4.4 Timed Behavior

Orc is a timed theory. Therefore, a faithful implementation of Orc requires capturing its timed behaviors. The notion of time in a language implementation is typically captured by a clock against which events in a program in that language may take place. There are several different ways in which clocks can be used to maintain timing information. For our distributed implementation, however, a number of requirements influence the design choices we have made. First, Orc's communication model is asynchronous. This suggests the use of *distributed clocks* (as opposed to a centralized clock), where each node in the distributed configuration maintains its local clock. Indeed, the distributed clocks architecture emphasizes Orc's philosophy of having the communicating expressions and sites as loosely coupled as possible. Furthermore, for all the applications we have so far specified in Orc, distributed clock synchronization is not required for preserving program correctness. This is primarily due to the fact that in most of the applications clocking information is used either locally (for example with the local *rtimer* site) or to time incoming responses. This greatly simplifies the implementation, since no clock synchronization mechanism, such as Lamport counters [24] or vector clocks [25, 26], is needed. Finally, since the implementation supports communication using sockets with external objects, which is inherently unpredictable given the possible transmission delays and network failures, any design of a clocking mechanism that depends on external communication with expressions or sites would also be unpredictable and unreliable.

Therefore, in Dist-Orc, for each node in the distributed configuration, the local clock is managed by an independent and local *ticker* object with access to the node's real-time system clock. Since in Maude there is no direct support for accessing the system clock, we employ sockets as a means of transmitting clock time ticks to Maude. It is important to note here that, although we use sockets to implement it, the ticker object is *local* to its corresponding Orc configuration and is thus guaranteed to provide fairly accurate clocking information. Figure 4 illustrates schematically the deployment architecture of a distributed Orc configuration with timing.

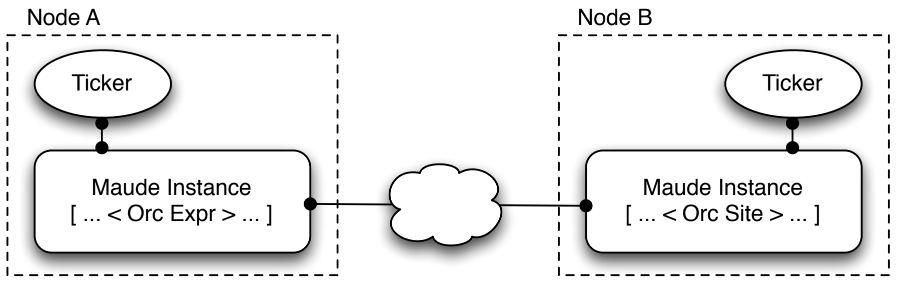
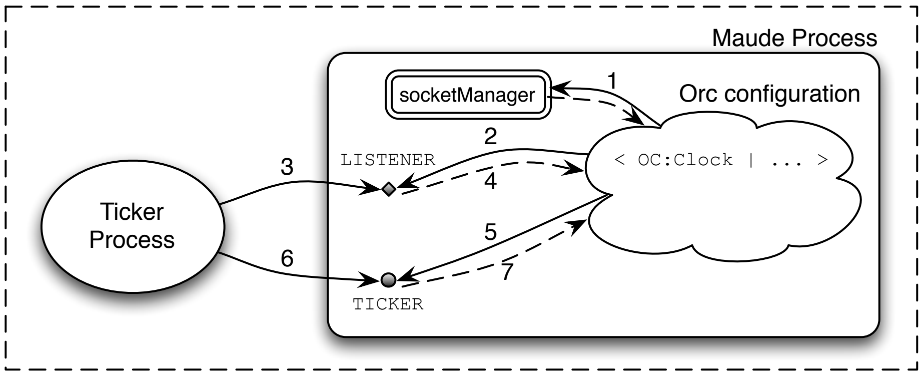


Fig. 4. A schematic diagram illustrating the general structure of a distributed Orc configuration. Dashed rectangles represent node boundaries, and darkened circles represent endpoints of TCP sockets.

The diagram in Figure 5 outlines the steps involved in initializing a connection with the co-located ticker object and receiving the first clock tick.

More specifically, upon initialization, the clock object within an Orc configuration requests a server socket to be created by the Maude socket manager, by issuing the message `createServerTcpSocket(...)`. Once the socket is created, the clock object then uses this socket to wait for an incoming connection from the local Ticker object, which is a Java process that is run in every node of a distributed configuration.



1: `createServerTcpSocket(socketManager, OC, PORT, ...)`
- creates the server socket `LISTENER`,
- elicits `createdSocket(OC, socketManager, LISTENER)`

2: `acceptClient(LISTENER, OC)`

3: Ticker requests a connection through `LISTENER`

4: `acceptedClient(OC, LISTENER, IP, TICKER)`
- creates the client socket `TICKER`

5: `receive(TICKER, OC)`

6: Ticker sends a tick string `S` to `TICKER`

7: `received(OC, TICKER, S)`

Fig. 5. The mechanics of establishing a connection with the ticker object and receiving clock ticks.

```
  rl [InitClockSocket1] :
     < OC : Clock | AS > createdSocket(OC, socketManager, LISTENER)
     => < OC : Clock | AS > acceptClient(LISTENER, OC) .
```

A Ticker object uses the built-in Java utility class `Timer` and the Java networking class `Socket` to generate and send a tick message every *t* milliseconds to its corresponding Maude process, where *t* is a positive integer value. Once the Ticker object establishes a connection with the clock object, the latter becomes ready to listen for incoming clock ticks.

```
  rl [InitClockSocket2] :
     < OC : Clock | clk: c(n) , AS >
     acceptedClient(OC, LISTENER, IP, TICKER)
     => < OC : Clock | clk: c(n) , AS > receive(TICKER, OC) .
```

Upon receiving a clock tick, the clock object updates its clock and reflects the effect of time elapse on the rest of the Orc configuration equationally using the time-updating function `delta`, which decrements the relative time delays in pending messages (see [18] for an explanation of the *delta* methodology).

```
crl [IncomingTick] :
  [ CF < OC : Clock | clk : c(n) , AS > received(OC, TICKER, S) ]
  => [ delta(CF) < OC : Clock | clk : c(s(n)) , AS > receive(TICKER, OC) ]
       if find(S, "#", 0) =/= notFound .
```

Recall that the operator `[ CF ]` encapsulates the localized configuration `CF`. The equational condition in the above tick rule checks whether the tick message has been fully received as buffering is (again) required to ensure proper message transmission (which is specified by a different rule similar to the rule `[AccumulateRequest]` shown in Section 4.2 for sites). The process of receiving and processing time tick messages keeps repeating as long as the Ticker object is supplying those ticks through the clock socket.

An important observation here is that the use of real, wall clock time in Dist-Orc to time Orc transitions eliminates the possibility of Zeno behaviors, which are a well-known artifact of logical time. This implies that for the intended semantics to be preserved, and hence correctness of the analysis later in Section 5, the transitions internal to an Orc configuration must be completed before the next real-time clock tick arrives. In other words, a single clock tick should be long enough to accommodate the instantaneous transitions of an Orc configuration. The minimum length of a clock tick so that this property is satisfied is specific to the Orc application and the machines used to run it. For example, for the distributed auction case study below, and using a 2.0GHz dual-core node with 4GB of memory, the clock tick can be made as short as 0.2 seconds. In general, deciding on a minimum size for a clock tick given an application is hard to anticipate and is typically accomplished through experimentation. Normally, for distributed Orc applications, it is enough to make sure that the application is designed so that a one-second clock tick is long enough for the application.

4.5 `Dist-Auction`: A Distributed Implementation of Auction

To illustrate Dist-Orc, we describe a distributed implementation `Dist-Auction` of the online auction management application in Orc, Auction, which was introduced in Section 2. The distributed configuration of the auction application contains two expression configurations: one with the *Posting* expression object, which is responsible for retrieving and posting items for sale by a given seller, and the other contains the *Bidding* expression object for managing the bidding process. For instance, the initial localized configuration for the *Posting* expression object has the form:

```
[<> < C : Clock | clk : c(0) >
 createServerTcpSocket(socketManager, C, 54200, 10)
 < E(l("10.0.0.2", 44200), 0) : Expr |
   env: Posting s := s("postNext") > x > AUCTIONID("post",x) >> rtimer(1)
                    >> Posting(s),
   exp: Posting(SELLERID), ... >  ...]
```

where `SELLERID` and `AUCTIONID` are object identifiers for the *Seller* and *Auction* sites, respectively. The *Posting* expression declaration is stored in the environment attribute `env` of the expression object, while the attribute `exp` keeps the actual expression to be evaluated. The configuration also includes objects for internal (fundamental) sites, such as *if* and *let*, which are omitted here for brevity.

In addition to the *Posting* and *Bidding* expression configurations, there are four site object configurations in the distributed configuration of `Dist-Auction`, one configuration for each of the sites assumed by AUCTION, namely *Seller*, *Bidders*, *MaxBid*, and *Auction*. For example, the initial localized configuration for a *Seller* site with two items for auction (identified by numbers `1910` and `1720`) may have the form:

```
[<> < C : Clock | clk : c(0) >
 createServerTcpSocket(socketManager, C, 54800, 10)
 < S(l("10.0.0.4", 44800), 0) : Site |
  name : 'seller, state : (item(1910, 5, 500), item(1720, 7, 700)) , ...>
 createServerTcpSocket(socketManager,
                                  S(l("10.0.0.4", 44800), 0), 44800, 10)]
```

We note that the site attempts to create two server sockets: one for listening to expression object requests and the other for listening to the local ticker object.

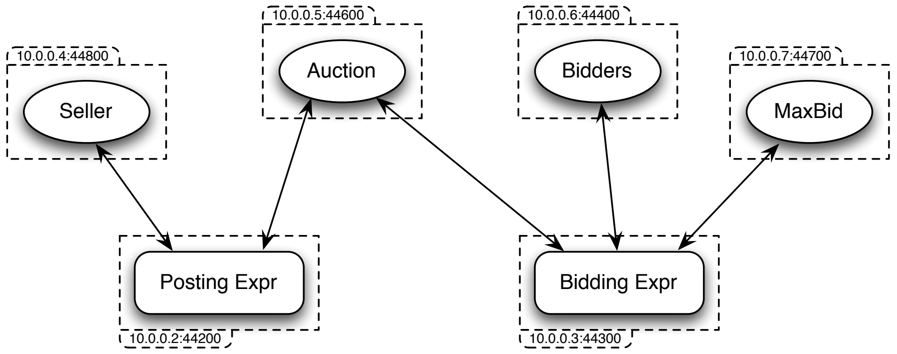


Fig. 6. The deployment architecture of the `Dist-Auction` Orc program

Each localized configuration in `Dist-Auction` may be run on a different node in a communication network. The diagram in Figure 6 depicts a physical deployment of `Dist-Auction`, with bidirectional arrows representing communication patterns. A physical deployment can be conveniently achieved using an appropriate shell script to run Maude, load the DIST-ORC module and `Dist-Auction`, and execute the external rewrite command `erew`. For example, with `initAuction` an operator that creates the initial state of the Auction site configuration, the following command executes the Auction site:

```
echo "erew initAuction ." | maude dist-orc.maude auction-manager.maude
```

with the following sample output, generated by the `print` attribute of Maude statements (with auction items `1910` and `1720`):

```
erewrite in DIST-AUCTION : initAuction .
Site "10.0.0.5":44600 0 initializing... Site is ready.
Clock server socket created.
Awaiting connection from ticker ...
Ticker connected.
Received "post"
Item 1910 posted
Received "getNext"
Bidding to start for 1910
Tick! ...  (5 time ticks)
Received "won"
Item 1910 won by Bidder 3
Received "getNext"
Tick!
Received "post"
Item 1720 posted
Bidding to start for 1720
Tick! ...  (6 time ticks)
Received "won"
Item 1720 won by Bidder 3
Received "getNext"
...
```

In this particular run, the auction site receives a `post` request from the *Posting* expression object and posts item `1910`. Meanwhile, a request for the next item to be auction is received from the *Bidding* expression object. The auction site then publishes the item details back to the *Bidding* expression, which takes care of orchestrating the bidding process for this item. After five time units (the duration of the auction on item `1910`), Bidder 3 is announced as the winner and a similar process is repeated for the second item `1720`.

5 Formal Analysis of Distributed Orc Programs

The real-time, distributed implementation of Orc described above is very useful in prototyping and deploying Orc programs on physically distributed nodes in an interconnected network. As we saw in Section 4.5, the implementation enables observing actual possible behaviors in practical environments, in which the effects of physical limitations of communication networks are taken into account.

However, the implementation technique as outlined above does not result in a language specification that is immediately amenable to more rigorous formal analysis such as reachability analysis and model-checking. This is fundamentally due to the fact that the implementation technique makes use of facilities that are outside the scope of the Maude formal analysis tools. In particular, there are two fundamental facilities in the implementation that complicate formal analysis: TCP sockets and the ticker objects. While support for sockets is built into Maude, sockets do not have a logical representation that can be subjected to formal analysis. Furthermore, the ticker objects, being written in another general-purpose language with access to the system's real, wall-clock time, introduce yet another obstacle in achieving a formally analyzable specification.

Our solution to this problem, which we explain in some detail in the rest of this section, is to come up with rewriting logic specifications for these facilities so that the distributed implementation can be turned, with minimal effort, into a formally analyzable specification in Maude. In particular, both Maude sockets and externally defined configuration clocks must be formally modeled at the object-level. This is discussed next.

5.1 Formal Specification of TCP Sockets

Maude's TCP sockets can be formally specified by building abstractions of Maude instances, sockets and their behaviors. We develop a rewriting specification $\mathcal{R}_{Socket}$ of sockets, which is based on previous related work on Mobile Maude [23, 12] and algorithmic skeletons in Maude [22]. The specification models sockets as a rewrite theory $\mathcal{R}_{Socket}$, in which Maude instances are abstracted with objects of the class *Process*, and server and client sockets as objects of *ServerSocket* and *Socket* classes, respectively. Abstract processes and sockets in $\mathcal{R}_{Socket}$ introduce a higher layer of abstraction in which socket objects mediate communication between processes, each of which encapsulates an Orc configuration as an attribute.

More specifically, a process object has the form $\langle PID : Process \mid sys : S\rangle$, with PID an object identifier and S an Orc configuration. A client socket object of the form

$$\langle SID : Socket \mid endpoints : [PID_1, PID_2]\rangle$$

abstracts a bidirectional client TCP socket set up between processes PID_1 and PID_2 (where $[PID_1, PID_2]$ is an unordered pair), while a server socket has the form:

$$\langle SID : ServerSocket \mid address : A, port : N, backlog : K\rangle$$

where K is a positive integer specifying the maximum allowed number of queue requests. While client sockets are created and destroyed during the course of execution of the different configurations, server sockets are created for server objects using a *Manager object*, which abstracts Maude's socket manager (which we first encountered in Section 4.2). The manager object itself is a simple object maintaining a counter for creation of fresh socket object identifiers.

To maximize specification modularity and reusability, socket objects in $\mathcal{R}_{Socket}$ have interfaces (i.e. message formats) that are almost identical to those of Maude sockets. This minimizes the need for making any changes in the distributed implementation of Orc when switching between Maude sockets and their abstractions given by $\mathcal{R}_{Socket}$.

The flexibility of the underlying rewriting logic on which the implementation is built allows for different possible abstraction levels at which the theory $\mathcal{R}_{Socket}$ can be specified. Indeed, by properly using equations and rules, the behavioral abstraction of sockets can be adjusted to match the desired abstraction level. On one extreme, the highest level of abstraction can be achieved by assuming a completely deterministic and predictable socket-based message exchange in which inter-process communication is essentially problem-free. In this case, $\mathcal{R}_{Socket}$ becomes a rewrite theory with no rewrite rules (an equational theory) $\mathcal{R}_{Socket} = (\Sigma_{Socket}, E_{Socket}, \emptyset)$. At the other extreme, the lowest level of abstraction can be achieved by assuming that all communication problems are possible. These potential problems include server and client socket creation and teardown failures, communication delays, and message transmission failures. In this case, all socket behaviors have now to be modeled with rewrite rules. By properly combining equations and rules, various useful abstraction levels in between these two extremes can be specified.

Although having a lower-level abstraction for sockets exposes more behaviors and enables more expressive reasoning by model-checking, the resulting specification is typically more non-deterministic and more susceptible to an exponential state space blow-up, which limits our ability to apply various formal analysis tools. Therefore, the abstraction level we have chosen for $\mathcal{R}_{Socket}$ and present below attempts to strike a balance between expressible properties and efficiency of checking them by essentially considering potential client socket creation errors and a somewhat limited form of communication delays and failures, while ignoring other possible errors related to server socket creation and teardown. The rationale behind this design choice is twofold. First, server socket creation and maintenance is generally more of an issue at a lower level (the underlying operating system level), and can be abstracted away in our formal analysis framework. Second, the way we capture client socket creation errors and delays in message delivery can be seen as an abstraction of actual messaging problems (unavailable or unreachable servers, and communication delays and drops through a lossy network channel). In this respect, the abstraction is sufficiently capable of expressing a wide range of properties related to these problems for external site calls and returns. A more detailed model of socket behaviors would severely limit our ability to apply the exhaustive formal analysis techniques for which this abstraction is built. In fact, if we were to focus on more concrete models of lossy communication, then other forms of formal analysis, such as statistical model checking and quantitative analysis, would be more suitable for analyzing lower-level behaviors of lossy channels since, in practice, delays and drop rates typically follow probabilistic distributions with very low probabilities of failure that cannot be satisfactorily modeled as standard rewrite rules, but can be modeled well as probabilistic extensions of them. Our aim in this work is to be able to efficiently apply exhaustive formal verification of various properties of distributed Orc programs. We discuss the main rules and equations specifying the abstracted behavior of sockets in $\mathcal{R}_{Socket}$.

Server socket creation is straightforward, and is abstracted with the following rewrite rule:

```
rl [CreateServerTcpSocket] :
  < PID : Process |
     sys : [createServerTcpSocket(socketManager, O, PORT, BACKLOG) CONF] >
  < socketManager : Manager | counter : N >
  => < PID : Process |
         sys : [createdSocket(O, socketManager, server(N)) CONF] >
     < socketManager : Manager | counter : (N + 1) >
     < server(N) : ServerSocket |
           address : "localhost", port : PORT, backlog : BACKLOG > .
```

The rule creates a server socket object with an arbitrary address and a given port, and transforms the socket creation request message into an appropriate response.

When an Orc object within a process attempts to create a client socket to a server by issuing the message `createClientTcpSocket (socketManager, O',` ADDRESS, PORT), two different transitions are possible depending on whether the client socket creation is successful or not. The success case is modeled by the following rule:

```
rl [CreateClientSocketSuccess] :
  < PID : Process | sys : [acceptClient(server(N), O) CONF] >
  < PID' : Process |
      sys : [createClientTcpSocket(socketManager, O', ADDRESS, PORT)
             CONF' ] >
  < socketManager : Manager | counter : M >
  < server(N) : ServerSocket | address : ADDRESS, port : PORT >
  => < PID : Process |
         sys : [acceptedClient(O, server(N), ADDRESS, socket(M)) CONF] >
     < PID' : Process |
         sys : [createdSocket(O', socketManager, socket(M))
                CONF'] >
     < socketManager : Manager | counter : (M + 1) >
     < server(N) : ServerSocket | address : ADDRESS, port : PORT >
     < socket(M) : Socket | endpoints : [PID : PID'] > .
```

In the rule above, the server is in a state accepting incoming connections from clients, which is specified in the rule by matching a server at address and port ADDRESS:PORT that is accepting connections using the message acceptClient. The rule also creates a socket object socket(M) that will mediate communication between the client and the server.

Client socket creation may also fail, representing situations where the server is unreachable or unavailable. This case is modeled by a similar rule, labeled [CreateClientSocketFail], with the same starting state as the rule above but with a different resulting state, where now the client process gets the socketError (O', socketManager, "") message from the socket manager, and no new socket object is created (see Appendix B.3).

Once a socket is successfully created, a connection through this socket is established, and bidirectional message exchanges may take place using send(...) and receive(...) messages. The following rule specifies message exchange between two processes:

```
crl [exchange] :
   < PID : Process | sys : [send(SOCKET, O, C) CONF] >
   < PID' : Process | sys : [receive(SOCKET, O') CONF'] >
   < SOCKET : Socket | endpoints : [PID : PID'] >
   < DID : Delays | ds : DS >
   => < PID : Process | sys : [sent(O, SOCKET) CONF] >
      < PID' : Process | sys : [received(O', SOCKET, C, R) CONF'] >
      < SOCKET : Socket | endpoints : [PID : PID'] >
      < DID : Delays | ds : DS >
     if DS' R DS'' := DS .
```

The send(...) and receive(...) messages are respectively transformed into a sent(...) message, acknowledging the send action to the sender, and a `received(O', SOCKET, C, R)` message, which signals (delayed) delivery of the sent message to the receiver *in `R` time units*. That is, the value(s) sent, `C`, are *delayed* by some amount of time `R` and will only be available to the receiver object after `R` time units have elapsed. The delay value for any transmitted message is non-deterministically extracted using a matching equation in the condition from a non-empty, set of delays `DS` maintained by a special object `< DID : Delays | ds : DS >` in the global configuration. To maintain feasibility of exhaustive formal analysis techniques, the set `DS` should obviously be finite. In fact, for most reasonably sized distributed Orc programs, the delay set should have a fairly small size. An appropriate delay set for a given distributed Orc application can be specified as part of its initial state using the `Delays` object. It is worth noting here that different behaviors may result by giving different delay sets. Two special cases of interest are: (1) $DS = \{0\}$, in which case messages are assumed to experience no delays, and (2) $\infty \in DS$, which represents the case of a lossy communication channel. As we will see in Section 5.2 below, the real-time semantics of the model will eventually make such delayed messages available to the receiver for processing.

Finally, closing a socket is straightforwardly modeled by the following equation:

```
eq [close] :
 < PID : Process | sys : [closeSocket(SOCKET, O) CONF] >
 < PID' : Process | sys : [receive(SOCKET, O') CONF'] >
 < SOCKET : Socket | endpoints : [PID : PID'] >
 = < PID : Process | sys : [closedSocket(O, socketManager, "") CONF] >
   < PID' : Process | sys : [closedSocket(O', socketManager, "") CONF'] > .
```

The equation drops the closed socket, and issues the `closedSocket(...)` message to its endpoints.

5.2 Global Logical Time

To enable formal analysis of timed behaviors of a distributed Orc configuration, time and its effects need to be formally specified. This is achieved by using the standard and general technique of capturing logical time in real-time rewrite theories [27], which is similar to the approach followed in our original semantics of Orc. Essentially, the time domain is representing by a sort *TimeInf* (the time with infinity), and a global *tick* rewrite rule is used to synchronously advance time and propagate its effects across the *encapsulated* global configuration. This technique is facilitated by Real-Time Maude [21], an extension of Maude that provides an implementation of real-time rewrite theories.

In particular, the distributed Orc configuration is encapsulated into a centralized *global configuration*, which is a term of the sort *GlobalSystem* and of the form $\{Conf\}$, where *Conf* is the Orc configuration consisting of all process and socket objects. The global configuration $\{Conf\}$ represents the state of the distributed Orc configuration at a given point in time. Furthermore, The tick rule, which plays the role of the ticker objects in the distributed implementation, is defined globally (on terms of *GlobalSystem*) as follows:

```
crl [tick] :
  { Conf } => { delta(Conf, R') } in time R'
    if eagerEnabled( { Conf } ) =/= true
    /\ R' <= mte(Conf) [nonexec] .
```

The tick rule computes on the global Orc configuration the function `delta`, which advances time for all local clock objects and updates time delays in all site calls and returns present in the configuration. For example, clocks and delayed external messages are updated, respectively, by the following two equations (`plus` and `monus` define addition and subtraction on time domains):

```
eq delta(< OC : Clock | clk : c(R) > CF, R')
   = < OC : Clock | clk : c(R plus R') > delta(CF, R') .
eq delta(received(O, O', C, R) CF, R')
   = received(O, O', C, R monus R') delta(CF, R') .
```

The tick rule above is not immediately executable (which is indicated by the `[nonexec]` attribute), as it introduces a new variable `R'` representing the amount of time elapse on its right hand side. A strategy for sampling time needs to be specified for the rule to be executable. In this case, we assume a general *maximal* strategy that in each tick advances time all the way to the next instant when an event could be triggered. The *maximum time elapse* for a tick transition is defined by the function `mte` as the minimum delay across all site call messages and returns in the global Orc configuration. The combination of the maximal time sampling strategy and the condition `R' <= mte(Conf)` in the tick rule ensures that time is advanced as much as possible in every tick but only enough to be able to capture all events of interest.

To properly capture the synchronous semantics of Orc, the tick rule is also made conditional on the fact that no other behavioral (instantaneous) transition is possible. The `eagerEnabled` predicate is true on global Orc configurations *Conf* that satisfy at least one of the following conditions:

1. *Conf* contains a process with an Orc expression object whose underlying Orc expression is active (has a site call, an expression call, and/or a publish expression that can be performed)
2. *Conf* contains a process with an Orc expression object and a site return to that object that can be (immediately) consumed
3. *Conf* contains two processes with objects that can (immediately) initiate a client socket connection or a message exchange

If *Conf* satisfies any one of the conditions above, it is said to be *eager*, and the tick rule cannot be applied. This imposes a precedence of rule application, where time ticks have the lowest priority among all instantaneous transitions, including internal transitions within the configuration and socket-based communication.

As was discussed in our earlier work [9, 10], the condition is necessary to precisely capture the intended semantics of the Orc theory.

It is important to note here that the abstraction of time and how it affects the global Orc configuration as specified by the tick rule is consistent with the real-time distributed implementation DIST-ORC in that, in DIST-ORC, we assumed that the granularity of a single time tick in real-time is always large enough for instantaneous transitions within a configuration to complete. Furthermore, the tick rule *synchronously* updates all clock objects in all processes. This also defines yet another abstraction over DIST-ORC, where individual clocks are not necessarily synchronized. However, since clock synchronization is not required for DIST-ORC, as was discussed in Section 4.4, the abstraction considers only those behaviors in DIST-ORC that make sense under these assumptions about time.

5.3 Further Abstractions For Performance

Unlike the formal specifications of sockets and time described above, the abstractions outlined below are not essential for formal reasoning about distributed Orc programs. They describe further optional abstractions that are useful for obtaining a more efficiently executable specification without affecting the kinds of properties that one would want to verify about Orc programs. The first optimization is to drop the meta-level operations in the definition of external communication between Orc objects across different processes, and define socket-based messaging at the level of Orc constants rather than at the lower-level of strings. This results in a higher abstraction that does not have to deal with serialization and de-serialization of Orc constants, as was required in the implementation DIST-ORC (see Section 4.3). It also entails a slight modification to the syntax of the socket specification in $\mathcal{R}_{socket}$. In particular, `send` and `received` messages each now take a list of Orc constants rather than a string as a parameter:

```
op send : Oid Oid ConstList -> Msg [ctor msg] .
op received : Oid Oid ConstList Time -> Msg [ctor msg] .
```

The rules and equations defining external, socket-based message exchange are also appropriately updated.

Another optimization, which aims at reducing the reachable state space of a distributed Orc program without changing the semantics of the underlying Orc expressions and sites, is to impose a slightly more restrictive rule application strategy. More specifically, we may give internal transitions of an Orc expression (site calls, expression calls, and publishing of values) priority over socket-based transitions (creating sockets, and sending and receiving external messages). Besides being natural, this strategy does not conflict with Orc's synchronous semantics, as internal transitions still have precedence over the external transition of consuming a site return. Furthermore, it turns out that this extension can be easily specified by simply changing the relevant rewrite rules in $\mathcal{R}_{socket}$ so that the underlying Orc expression objects have expressions that are *inactive*.

For instance, here is a fragment of the modified rule specifying successful client socket creation `[CreateClientSocketSuccess]`. The rule matches an inactive expression `iF` in the Orc expression object trying to make a connection with an external site:

```
rl [CreateClientSocketSuccess] :
  < PID : Process | sys : [acceptClient(server(N), O) CONF] >
  < PID' : Process |
     sys : [createClientTcpSocket(socketManager, P(OE,h), ADDRESS, PORT)
           < P(OE,h) : Proxy | > < OE : Expr | exp : iF > CONF' ] >
  < socketManager ... > < server(N) ... >
  => < PID : Process |
        sys : [acceptedClient(O, server(N), ADDRESS, socket(M)) CONF] >
     < PID' : Process |
        sys : [createdSocket(P(OE,h), socketManager, socket(M))
              < P(OE,h) : Proxy | > < OE : Expr | exp : iF > CONF'] >
     < socketManager ... > < server(N) ... > < socket(M) ... > .
```

The full set of updated rules can be found in Appendix B.3.

5.4 Formal Analysis of `Dist-Auction`

The formal specification of sockets and logical time provides a formal model of Dist-Orc that can be used to verify various properties about distributed applications in Orc. To illustrate how Dist-Orc programs can thus be analyzed, we use some of the formal tools provided by Real-Time Maude to verify some properties about the distributed implementation `Dist-Auction` of the auction application, Auction, introduced in Section 4.5. For our analysis, we assume a single seller site with two items for sale, labeled `1910` and `1720`, and offered for auction for 5 and 7 time unites respectively.

The properties are specified by first defining the required set of atomic predicates, which are: (1) *commError*, which is true in states with communication errors, (2) $hasBid(id)$, with id an item identifier, (3) $sold(id)$, (3) $conflict(id)$, which is true in states having two distinct winners for the item id, and (5) $max(id)$, which is true in states where id is sold to the highest bidder. The operator $initial(DS)$ constructs an initial state for Dist-Auction in which the set of possible message transmission delays is DS, which, in the analysis examples below, is assumed to be the singlton set $\{0.1\}$, unless otherwise indicated.

A property that is typically required in an auction management system is that an item with at least one bid is eventually sold: $\Box \bigwedge_i (hasbid(id_i) \rightarrow \Diamond sold(id_i))$. This can be shown to be guaranteed by Dist-Auction in the absence of communication problems and excessively large delays:

```
Maude> (mc initial(1/10) |=t commitAllNoErrors in time <= 15 .)
rewrites: 7052663 in 14413ms cpu (14420ms real) (489317 rewrites/second)
Model check initial(1/10) |=t commitAllNoErrors in AUCTION-MODEL-CHECK
  in time <= 15 with mode maximal time increase
Result Bool : true
```

The operator `commitAllNoErrors` is a formula that specifies the property above when no socket-related errors are allowed. The property is satisfied with a communication delay of 0.1 time units. In fact, the property is satisfied when communication delays are bounded above by 0.25 time units. This is because the timeout value for collecting bids in a single bidding round in the *TimeoutRound* expression is 1.0, while a delay of 0.25 translates into a cumulative round trip delay of 1.0 for its two sequential site calls, which may result in an uncommitted bid. This can be verified by the resulting counterexample when executing the command above with `initial(1/4)`.

An auction management system must guarantee that every item sold has a unique winner: $\Box \bigwedge_i \neg \mathit{conflict}(id_i)$. This property is in fact satisfied in Dist-Auction regardless of communication errors and delays, as can be seen by executing the command (with `uniqueWinnerAll` denoting the formula above).

```
Maude> (mc initial(1/10) |=t uniqueWinnerAll in time <= 15 .)
rewrites: 8613539 in 19627ms cpu (19800ms real) (438843 rewrites/second)
Model check initial(0) |=t uniqueWinnerAll in AUCTION-MODEL-CHECK
  in time <= 15 with mode maximal time increase
Result Bool : true
```

Another property of interest is that when an item is sold, then it must have been sold at the highest bid: $\Box \bigwedge_i (\mathit{sold}(id_i) \rightarrow \mathit{max}(id_i))$. This property is satisfied by Dist-Auction under the condition that delays are diffirent from 0.25.

```
Maude> (mc initial(1/10) |=t winmaxAll in time <= 15 .)
rewrites: 6969457 in 14313ms cpu (14331ms real) (486916 rewrites/second)
Model check initial(1/10) |=t winmaxAll in AUCTION-MODEL-CHECK
  in time <= 15 with mode maximal time increase
Result Bool : true
```

The property is clearly satisfied when the delays are less than 0.25. However, delays larger than 0.25 may result in no item being won, and thus the property vacuously holds. This can be verified by running the command above with `initial(1/4)`.

Finally, given a delay of 0.1, one can verify that the first item cannot be won before 5.5 time units have passed using the following command:

```
Maude> (find earliest initial(1/10) =>* {C:Configuration}
                 such that {C:Configuration} |= sold(1910) .)
rewrites: 268287407 in 1525921ms cpu (1544117ms real)
    (175819 rewrites/second)
Find earliest {C:Configuration} in AUCTION-MODEL-CHECK such that
initial(1/10)=>* {C:Configuration}
with mode maximal time increase :

Result: {< did : Delays | ds : 1/10 > ... } in time 11/2
```

This can also be observed by careful examination of the *Bidding* expression.

6 Conclusion and Future Work

We have presented Dist-Orc, a rewriting-based, real-time, distributed implementation of the Orc language allowing different Orc expressions at different locations to interact by asynchronous message passing with different sites in an object-based manner. We have also shown how a Dist-Orc real-time implementation can be easily obtained from a rewriting logic semantic definition of Orc in a correct-by-construction way using Maude sockets and ticker objects. And we finally demonstrated with an auction example that Orc applications running in Dist-Orc can still be formally analyzed by model checking once we model the distributed infrastructure at a reasonable level of abstraction.

Much work remains ahead. Besides developing a broader class of examples and optimizing the present prototype, three interesting future directions are: (i) developing a transformation method based on the techniques presented here that can automatically synthesize a real-time, distributed implementation from the formal semantics; (ii) making Dist-Orc more user-friendly, by providing a user interface for interacting with Dist-Orc-based web orchestration applications; and (iii) endowing Dist-Orc with a *security infrastructure*, and formally verifying the security of certain types of web orchestration services that use such an infrastructure against general classes of attacks.

Acknowledgements. We thank Jayadev Misra, William Cook, David Kitchin, Adrian Quark, and Andrew Matsuoka for very fruitful discussions on Orc and its Java implementation. This work has been partially supported by NSF grant NSF CNS 07-16638, King Fahd University of Petroleum and Minerals, and King Abdullah University of Science and Technology.

References

1. Misra, J.: Computation orchestration: A basis for wide-area computing. In Broy, M., ed.: Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems. NATO ASI Series, Marktoberdorf, Germany (2004)
2. Misra, J., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Journal of Software and Systems Modeling **6**(1) (March 2007) 83–110
3. Hoare, T., Menzel, G., Misra, J.: A tree semantics of an orchestration language. Engineering Theories of Software Intensive Systems (2005) 331–350
4. Kitchin, D., Cook, W.R., Misra, J.: A language for task orchestration and its semantic properties. In: CONCUR 2006. Volume 4137 of Lecture Notes in Computer Science., Springer (2006) 477–491
5. Rosario, S., Kitchin, D., Benveniste, A., Cook, W., Haar, S., Jard, C.: Event structure semantics of Orc. In: WS-FM 2007. Volume 4937 of Lecture Notes in Computer Science., Springer (2008) 154–168
6. Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: A timed semantics of Orc. Theor. Comput. Sci. **402**(2-3) (2008) 234–248
7. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc programming language. In Lee, D., Lopes, A., Poetzsch-Heffter, A., eds.: Formal techniques for Distributed

Systems; Proceedings of FMOODS/FORTE. Volume 5522 of LNCS., Springer (2009) 1–25

8. Cook, W.R., Misra, J.: Implementation outline of Orc. http://orc.csres.utexas.edu (2005)
9. AlTurki, M., Meseguer, J.: Real-time rewriting semantics of Orc. In: PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming, New York, NY, USA, ACM Press (2007) 131–142
10. AlTurki, M., Meseguer, J.: Reduction semantics and formal analysis of Orc programs. Electron. Notes Theor. Comput. Sci. **200**(3) (2008) 25–41
11. AlTurki, M., Meseguer, J.: Rewriting logic semantics of Orc. Technical Report UIUCDCS-R-2007-2918, University of Illinois at Urbana Champaign (November 2007)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. Volume 4350 of LNCS. Springer-Verlag, Secaucus, NJ, USA (2007)
13. Sun, M., Meseguer, J.: Distributed real-time emulation of formally-defined patterns for safe medical device control. In: The 1st International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS), Longyearbyen, Spitsbergen, Norway (April 2010)
14. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1) (2003) 5–51
15. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1) (1992) 73–155
16. Meseguer, J., Roşu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: Proc. Intl. Joint Conf. on Automated Reasoning IJCAR'04, Cork, Ireland, July 2004, Springer LNAI 3097 (2004) 1–44
17. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3) (2007) 213–237
18. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2) (2007) 161–196
19. Meseguer, J., Braga, C.: Modular rewriting semantics of programming languages. Algebraic Methodology and Software Technology (2004) 364–378
20. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. Electron. Notes Theor. Comput. Sci. **176**(4) (2007) 5–27
21. Ölveczky, P.C.: Real-Time Maude 2.3 manual (August 2007) http://heim.ifi.uio.no/ peterol/RealTimeMaude/.
22. Riesco, A., Verdejo, A.: Distributed applications implemented in Maude with parameterized skeletons. In Bonsangue, M.M., Johnsen, E.B., eds.: FMOODS. Volume 4468 of Lecture Notes in Computer Science., Springer (2007) 91–106
23. Durán, F., Riesco, A., Verdejo, A.: A distributed implementation of Mobile Maude. Electron. Notes Theor. Comput. Sci. **176**(4) (2007) 113–131
24. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
25. Mattern, F.: Virtual time and global states of distributed systems. In et al., C.M., ed.: Proc. Workshop on Parallel and Distributed Algorithms, North-Holland / Elsevier (1989) 215–226 (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).
26. Fidge, C.J.: Timestamps in message-passing systems that preserve partial ordering. In: Proceedings of the 11th Australian Computer Science Conference. (February 1988) 56–66

27. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science **285** (August 2002) 359–405

A Orc's Distributed Implementation Dist-Orc in Maude

A.1 Orc Syntax and Explicit Substitution

```
fmod LOCATION is
  pr STRING .
  sort Loc .
  op l : String Nat -> Loc [ctor] .
endfm

mod OID is
  pr LOCATION .
  pr QID .
  ex CONFIGURATION .

  sorts ISOid XSOid SOid EOid .
  subsort ISOid XSOid < SOid .
  subsort SOid EOid < Oid .

  ops let if rtimer clock : -> ISOid [ctor] .
  op S : Qid -> ISOid [ctor] .
  op S : Loc Nat -> XSOid [ctor] .
  op E : Loc Nat -> EOid [ctor] .
endm

mod SYSTEM is
  inc CONFIGURATION .
  sort LocalSystem .
  op `[_`] : Configuration -> LocalSystem [ctor] .
endm

fmod VARIABLES is
  pr QID .
  sorts Var IVar .
  op v : Qid -> Var [ctor] .
  op _{_} : Var Nat -> IVar [ctor prec 1] .
endfm

fmod EXPR-NAMES is
  pr QID .
  sort ExprName .
  op e : Qid -> ExprName [ctor] .
endfm

mod CONSTANTS is
  inc OID .
  pr QID .

  sorts Const PreConst ConstList .
  subsort SOid String < Const .       ---- String: current encoding does not allow white spaces
                                      ---- use c(S) if S has to have white spaces
  subsort Const < PreConst .
  subsort Const < ConstList .

  op signal :           -> Const [ctor] .
  op b      : Bool      -> Const [ctor] .
  op i      : Nat       -> Const [ctor] .
  op l      : ConstList -> Const [ctor] .
  op c      : String    -> Const [ctor] .
endm

mod PARAMETER is
```

```
  pr VARIABLES .
  pr EXPR-NAMES .
  pr CONSTANTS .

  sorts AParam FParam AParamList FParamList .
  subsorts Var < FParam < FParamList .
  subsorts IVar 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] .
endm

fmod HANDLE is
  pr NAT .
  sort Handle .
  op h : Nat -> Handle [ctor] .
endfm

mod EXPRESSION is
  pr PARAMETER .
  pr HANDLE .
  pr CONVERSION .

  sorts AExpr IExpr Expr ExprList .
  subsort AExpr IExpr < Expr < ExprList .

  op nilE : -> ExprList [ctor] .
  op _,_ : ExprList ExprList -> ExprList [ctor assoc id: nilE prec 35] .

  op zero : -> IExpr [ctor] .
  op _(_) : SOid AParamList -> Expr  [ctor prec 10] .
  op _(_) : SOid ConstList ->  AExpr [ctor prec 10] .
  op _(_) : IVar AParamList -> IExpr [ctor prec 10] .

  op _(_) : ExprName AParamList -> AExpr [ctor prec 10] .

  op !_ : IVar  -> IExpr [ctor prec 5] .
  op !_ : Const -> AExpr [ctor prec 5] .

  op _>_>_ : Expr Var Expr -> Expr [ctor frozen (3) prec 15 gather (e & E)] .
  op _>_>_ : AExpr Var Expr -> AExpr [ditto] .
  op _>_>_ : IExpr Var Expr -> IExpr [ditto] .

  op _|_   : Expr  Expr  -> Expr [ctor assoc comm prec 20] .
  op _|_   : AExpr Expr  -> AExpr [ditto] .
  op _|_   : IExpr IExpr -> IExpr [ditto] .

  op _<_<_ : Expr Var Expr   -> Expr [ctor prec 25 gather (E & e)] .
  op _<_<_ : AExpr Var Expr  -> AExpr [ditto] .
  op _<_<_ : Expr Var AExpr  -> AExpr [ditto] .
  op _<_<_ : IExpr Var IExpr -> IExpr [ditto] .

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

  vars f g : Expr . vars fl : ExprList . vars E : ExprName .
  vars SO : SOid . vars AL AL' : AParamList .
  vars I : Nat . vars Q : Qid . vars IX : IVar .

  --- inactive site calls
  mb SO(AL, IX, AL') : IExpr .

  *** Syntactic sugar definitions
  op _() : SOid     -> Expr [prec 10] .
  eq SO() = SO(nilA) .

  op _() : IVar     -> Expr [prec 10] .
  eq IX() = IX(nilA) .

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

  op _>>_ : Expr Expr -> Expr [frozen (2) prec 15 gather (e E)] .
  eq f >> g = f > v('#!) > g .           --- assuming v('#) is never used in g

  op !_ : Expr  -> Expr [prec 5] .
  eq ! f = ! v('#1){0} < v('#1) < f .

  op _(_) : SOid ExprList   -> Expr [prec 10] .
  eq SO(nilE) = SO(nilA) .
  eq SO(f,fl) = desugar((f,fl), SO, nilA, 0) .

  op desugar : ExprList SOid AParamList Nat -> [Expr] .
 ceq desugar((f,fl), SO, AL, I) = f > v(Q) > desugar(fl, SO, (AL,v(Q){0}), s(I))
     if Q := qid("#" + string(I, 10)) .
  eq desugar(nilE, SO, AL, I) = SO(AL) .

  op _(_) : IVar ExprList   -> Expr [prec 10] .
  eq IX(nilE) = IX(nilA) .
  eq IX(f,fl) = desugar((f,fl), IX, nilA, 0) .

  op desugar : ExprList IVar AParamList Nat -> [Expr] .
 ceq desugar((f,fl), IX, AL, I) = f > v(Q) > desugar(fl, IX, (AL,v(Q){0}), s(I))
     if Q := qid("#" + string(I, 10)) .
  eq desugar(nilE, IX, AL, I) = IX(AL) .
endm

mod ORC-SYNTAX is
  inc EXPRESSION .

  sorts Prog Decl DeclList .
  subsort Decl < DeclList .

  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] .

  vars x : Var . vars f g : Expr . vars C : ConstList .
  vars E : ExprName . vars SO : SOid . vars ix : IVar .
  vars c c' : Const . vars n n' : Nat . vars b : Bool .

  eq zero > x > f = zero .
  eq zero | f = f .

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

mod ORC-EXTENDED-SYNTAX is
  pr ORC-SYNTAX .

  *** Internal constructs
  sort Builtin .
  op _(_) : Builtin AParamList -> Expr [prec 10] .
  op _(_) : Builtin ConstList -> AExpr [prec 10] .

  vars f : Expr . vars fl : ExprList . vars AL AL' : AParamList .
  vars I : Nat . vars Q : Qid . vars BI : Builtin . var IX : IVar .
  --- inactive builtin calls
  mb BI(AL, IX, AL') : IExpr .

  --- syntactic sugar
  op _(_) : Builtin ExprList   -> Expr [prec 10] .
  eq BI(nilE) = BI(nilA) .
  eq BI(f,fl) = desugar((f,fl), BI, nilA, 0) .

  op desugar : ExprList Builtin AParamList Nat -> [Expr] .
 ceq desugar((f,fl), BI, AL, I) = f > v(Q) > desugar(fl, BI, (AL,v(Q){0}), s(I))
     if Q := qid("#" + string(I, 10)) .
  eq desugar(nilE, BI, AL, I) = BI(AL) .


  ops empty head tail size append it : -> Builtin [ctor] .
  ops neg : -> Builtin [ctor] .
  ops eq lt gte : -> Builtin [ctor] .
  ops sub add : -> Builtin [ctor] .
  ops min max : -> Builtin [ctor] .

  vars C : ConstList .
  vars c c' : Const . vars n n' : Nat . var b : Bool .

  eq empty(l(nilA)) = ! b(true) .
  eq empty(l(c, C)) = ! b(false) .

  eq head(l(c, C)) = ! c .
  eq tail(l(c, C)) = ! l(C) .

  eq it(i(n), l(c, C)) = ! it*(n, l(c, C)) .
  op it* : Nat Const -> Const .
  eq it*(s(n), l(c, C)) = it*(n, l(C)) .
  eq it*(0, l(c, C)) = c .


  eq size(l(C)) = ! i(size*(l(C))) .
  op size* : Const -> Nat .
  eq size*(l(nilA)) = 0 .
  eq size*(l(c, C)) = s(size*(l(C))) .

  eq append(c, l(C)) = if (c == signal) then ! l(C) else ! l(c, C) fi .

  eq neg(b(b)) = ! b(not(b)) .

  eq eq(c, c') = ! b(c == c') .
  eq lt(i(n), i(n')) = ! b(n < n') .
  eq gte(i(n),i(n')) = ! b(n >= n') .

  eq add(i(n), i(n')) = ! i(n + n') .
  eq sub(i(n), i(n')) = ! i(pred(n, n')) .
  op pred : Nat Nat -> Nat .
  eq pred(s(n), s(n')) = pred(n, n') .
  eq pred(n, 0) = n .
  eq pred(0, n) = 0 .

  eq min(i(n), i(n')) = ! i(min(n, n')) .
  eq max(i(n), i(n')) = ! i(max(n, n')) .
endm


mod CINNI is
  pr 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 .
endm

mod SUBSTITUTION is
  pr CINNI .
  pr ORC-EXTENDED-SYNTAX .

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

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

  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 SO : SOid .
  var h : Handle . var u : Builtin .

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

  eq S zero = zero .
  eq S (SO ( P )) = SO ( S P ) .
  eq S (E ( P ))  = E ( S P ) .
  eq S (ix ( P )) = (S ix) ( 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 < x < f') = ([lift x S] f) < x < (S f') .
  eq S (? h) = ? h .

  eq S u(P) = u(S P) .

  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] .
endm
```

A.2 Orc Rewriting Semantics

```
fmod CLOCK is
  pr NAT .
  sort ClockAttr .
  op halt : -> ClockAttr [ctor] .
  op c : Nat -> ClockAttr [ctor] .
  eq c(30) = halt . --- finite state clock
endfm
```

```
mod HANDLE-SET is
  ex ORC-EXTENDED-SYNTAX .
  sort HandleSet .
  subsort Handle < HandleSet .

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

  var h : Handle . var H : HandleSet .
  eq h , h , H = h , H .
endm

mod HANDLE-OPS is
  pr HANDLE-SET .

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

  --- set membership
  op _in_ : Handle HandleSet -> Bool .
  eq h in (h , H) = true .
  eq h in H = false [owise] .

  --- generating fresh handle names wrt a given set
  op gFresh : HandleSet -> Handle .
  op gFreshAux : HandleSet Handle -> Handle .

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

  --- the set of handles occurring in an expression
  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 < x < f') = handles(f) , handles(f') .
  eq handles(zero)  = mth .
  eq handles(OS(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 .
endm

mod ENVIRONMENT is
  pr ORC-EXTENDED-SYNTAX .
  sort Env .
  subsort Decl < Env .

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

  var E : ExprName . var f f' : Expr .
  var sigma : Env . 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] .
```

```
endm


mod EVENT is
  pr ORC-EXTENDED-SYNTAX .

  sort Event .
  op [_:_@_] : Oid Const Nat -> Event [ctor] .
endm

mod TRACE is
  pr EVENT .
  sort PubTrace .
  subsort Event < PubTrace .
  op nil : -> PubTrace [ctor] .
  op _,_ : PubTrace PubTrace -> PubTrace [ctor assoc id: nil] .
endm


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

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

  op sr : PreConst Handle Nat -> Content [ctor] .          --- local site returns
  op sr : PreConst Oid Nat -> Content [ctor] .             --- external site returns
  op sc : Oid ConstList Handle Nat -> Content [ctor] .
endm

-------------------------------------------------------------------------------------
mod ORC-CLOCK is
  pr CLOCK .
  pr STRING .
  inc CONFIGURATION .

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

-------------------------------------------------------------------------------------
mod ORC-EXPR is
  pr HANDLE-OPS .
  pr ENVIRONMENT .

  op Expr : -> Cid [ctor] .

  op exp`:_ : Expr      -> Attribute [ctor gather (&) frozen] .
  op env`:_ : Env       -> Attribute [ctor gather (&)] .
  op hdl`:_ : HandleSet -> Attribute [ctor gather (&)] .

endm

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

  op Site : -> Cid [ctor] .

  sort Op .
  op free    : -> Op [ctor] .
  op exec    : ConstList Handle Oid -> Op [ctor] .  --- for internal sites
  op exec    : ConstList Oid -> Op [ctor] .         --- for external sites

  sort OState .
  op nil : -> OState [ctor] .
  sort OStatus .
  ops idle initializing active : -> OStatus [ctor] .

  --- for both internal and external sites
  op name`:_   : Qid -> Attribute [ctor gather (&)] .
  op op`:_     : Op  -> Attribute [ctor gather (&)] .
  --- for external sites only
  op state`:_  : OState  -> Attribute [ctor gather (&)] .
  op status`:_ : OStatus -> Attribute [ctor gather (&)] .
  op buffer`:_ : String  -> Attribute [ctor gather (&)] .
endm

-----------------------------------------------------------------------------------
mod ORC-PROXY is
  inc ORC-EXTENDED-SYNTAX .

  --- Proxy object
  op Proxy : -> Cid [ctor] .
  op P : EOid Handle -> Oid [ctor] .

  op param`:_    : ConstList  -> Attribute [ctor gather (&)] .
  op response`:_ : String -> Attribute [ctor gather (&)] .
endm

-----------------------------------------------------------------------------------
mod ORC-OBSERVER is
  inc TRACE .

  op PubTrace : -> Cid [ctor] .
  op T : -> Oid [ctor] .

  op t`:_ : PubTrace -> Attribute [ctor gather (&)] .
endm

mod MYRANDOM is
  pr RANDOM .
  pr COUNTER .

  op rand : -> [Nat] .
  eq rand = 0 .
endm

mod ORC-RWSEM is
  pr ORC-EXPR .
  pr ORC-SITE .
  pr ORC-CLOCK .
  pr ORC-MESSAGE .
  pr ORC-OBSERVER .
  pr SUBSTITUTION .
  pr CONVERSION .
  pr MYRANDOM .

  var OT OC : Oid . var OE : EOid .
  var iOS : ISOid . var xOS : XSOid . var OS : SOid .
  vars AS AS' : AttributeSet .
  vars CF CF' : Configuration .
  var d : Decl . var D : DeclList .
  var h h' : Handle . var H H' : HandleSet .
  var M : Qid .
  var c c' : Const .
  var x : Var . var ix : IVar .
  var P : AParamList . var C : ConstList .
  vars f f' g g' : Expr . vars iF : IExpr . vars aF aF' : AExpr .
  var n n' m : Nat .
  var E : ExprName . var Q : FParamList .
  var sigma : Env .
  var u : Builtin .
  var R : PubTrace .
```

```
 vars W W' : [Expr] .
 var pc : PreConst .

 --- a temporary handle expression
 op tmph : -> IExpr .
 eq S:Subst tmph = tmph .

 sort S$Const .        --- a special const sort used internally for performance optimization only
 subsort S$Const < PreConst .
 op $ : -> S$Const .

 --- predicates on expressions
 sort CallEventType .
 ops sc ec : -> CallEventType .
 op hasCall? : CallEventType Expr -> Bool [frozen] .
 op hasPub? : Expr -> Bool [frozen] .

 var e : CallEventType .

ceq hasCall?(e, f | f') = hasCall?(e, f) == true  or hasCall?(e, f') == true
       if f =/= zero /\ f' =/= zero .
 eq hasCall?(e, f > x > f') = hasCall?(e, f) .
 eq hasCall?(e, f < x < f') = hasCall?(e, f) == true or  hasCall?(e, f') == true .
 eq hasCall?(sc, OS(C)) = true .
 eq hasCall?(ec, E(P)) = true .
 eq hasCall?(e, f) = false [owise] .

ceq hasPub?(f | f') = hasPub?(f) == true  or hasPub?(f') == true
       if f =/= zero /\ f' =/= zero .
 eq hasPub?(f > x > f') = hasPub?(f) .
 eq hasPub?(f < x < f') = hasPub?(f) or hasPub?(f') .
 eq hasPub?(! c) = true .
 eq hasPub?(f) = false [owise] .

 ************ Rewrite rules
 *** SiteCall
 op scallup : Expr SOid ConstList -> [Expr] [frozen] .
 op scalldn : Expr Handle -> Expr [frozen (1)] .

crl [SiteCall] : < OE : Expr | exp : aF , AS > => < OE : Expr | exp : scallup(f', OS, C) , AS >
                       if hasCall?(sc, aF)
                       /\ aF => scallup(f', OS, C) .
 rl [SiteCall*] : OS(C) => scallup(tmph, OS, C) .

 *** SiteRet
 op sret : Expr PreConst Handle -> Expr [frozen(1)] .

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

 *** Pub
 op pub : Expr PreConst -> [Expr] [frozen] .

crl [Pub]    : < OE : Expr | exp : aF, AS > => < OE : Expr | exp : pub(f', pc) , AS >
                  if hasPub?(aF) /\ aF => pub(f', pc) .
 rl [Pub*] : ! c => pub(zero, c) .

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

crl [ExprCall] : < OE : Expr | exp : aF , AS > => < OE : Expr | exp : ecallup(f', E, P) , AS >
                   if hasCall?(ec, aF)
                   /\ aF => ecallup(f', E, P) .
 rl [ExprCall*] : E(P) => ecallup(tmph, E, P) .
 ************** Equations specifying how information is propagated

ceq scallup(f, OS, C) | f' = scallup(f | f', OS, C) if f' =/= zero .
 eq scallup(f, OS, C) > x > f' = scallup(f > x > f', OS, C) .
 eq scallup(f, OS, C) < x < f' = scallup(f < x < f', OS, C) .
 eq f' < x < scallup(f, OS, C) = scallup(f' < x < f, OS, C) .

ceq < OE : Expr | exp : scallup(f, OS, C) , hdl : H , AS >
    = < OE : Expr | exp : scalldn(f, h) , hdl : h , H , AS >
      OS <- 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 < x < f', h) = scalldn(f, h) < x < scalldn(f', h) .
 eq scalldn(zero, h) = zero .
 eq scalldn(OS(P), h) = OS(P) .
 eq scalldn(E(P), h) = E(P) .
 eq scalldn(u(P), h) = u(P) .
 eq scalldn(ix(P), h) = ix(P) .
 eq scalldn(! ix, h) = ! ix .
 eq scalldn(! c, h) = ! c .
 eq scalldn(? h', h) = ? h' .
 eq scalldn(tmph, h) = ? h .

eq OE <- sr($, h, 0) < OE : Expr | exp : iF , hdl : h , H >
    = < OE : Expr | exp : sret(iF, $, h) , hdl : H >
      .

ceq sret(f | f', pc, h) = sret(f, pc, h) | sret(f', pc, h)
       if f =/= zero /\ f' =/= zero .
 eq sret(f > x > f', pc, h) = sret(f, pc, h) > x > sret(f', pc, h) .
 eq sret(f < x < f', pc, h) = sret(f, pc, h) < x < sret(f', pc, h) .
 eq sret(zero, pc, h) = zero .
 eq sret(OS(P), pc, h) = OS(P) .
 eq sret(E(P), pc, h) = E(P) .
 eq sret(ix(P), pc, h) = ix(P) .
 eq sret(u(P), pc, h) = u(P) .
 eq sret(! ix, pc, h) = ! ix .
 eq sret(! c', pc, h) = ! c' .
 eq sret(? h(n'), c, h(n)) = if (n' == n) then ! c else ? h(n') fi .
 eq sret(? h(n'), $, h(n)) = if (n' == n) then zero else ? h(n') fi .

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

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

ceq pub(f, $) | f' = pub(f | f', $) if f' =/= zero .
 eq pub(f, $) > x > f' = pub(f > x > f', $) .
 eq pub(f, $) < x < f' = pub(f < x < f', $) .
 eq f' < x < pub(f, $) = pub(f' < x < f, $) .

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

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) < x < f' = ecallup(f < x < f', E, P) .
 eq f' < x < ecallup(f, E, P) = ecallup(f' < x < f, E, P) .

 eq < OE : Expr | exp : ecallup(f, E, P) , env : (sigma , E Q := g) , AS > =
      < OE : Expr | exp : ecalldn(f, ([Q <- P] g)) , env : (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 < x < f', g) = ecalldn(f, g) < x < ecalldn(f', g) .
  eq ecalldn(zero, g) = zero .
  eq ecalldn(OS(P), g) = OS(P) .
  eq ecalldn(E(P), g) = E(P) .
  eq ecalldn(u(P), g) = u(P) .
  eq ecalldn(ix(P), g) = ix(P) .
  eq ecalldn(! ix, g) = ! ix .
  eq ecalldn(! c, g) = ! c .
  eq ecalldn(? h, g) = ? h .
  eq ecalldn(tmph, g) = g .


  *** Internal Site consuming a call
  eq  iOS <- sc(OE, C, h, O) < iOS : Site | name : M , op : free, AS >
    = < iOS : Site | name : M , op : exec(C, h, OE) , AS > .

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

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

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

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

  ---- 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) < x < W = err if W :: Expr == false .
 ceq W < x < ecallup(f, E, P) = err if W :: Expr == false .

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

mod ORC-SITES is
  inc ORC-RWSEM .

  var OE : EOid . var OS : SOid . var iOS : SOid . vars 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 : Qid . 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 : Env .

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

  --- if
  eq < iOS : Site | name : 'if, op : exec(b(true), h, OE) , AS > =
       < iOS : Site | name : 'if , op : free , AS >
       OE <- sr(signal, h, rand) .

  eq < iOS : Site | name : 'if, op : exec(b(false), h, OE) , AS > =
       < iOS : Site | name : 'if , op : free , AS >
       OE <- sr($, h, 0)
       .

  --- clock
  eq < iOS : Site | name : 'clock, op : exec(nilA, h, OE) , AS > < OC : Clock | clk : c(t) > =
       < iOS : Site | name : 'clock , op : free , AS > < OC : Clock | clk : c(t) >
       OE <- sr(i(t), h, 0) .

  --- rtimer
  eq < iOS : Site | name : 'rtimer, op : exec(i(n), h, OE) , AS > =
       < iOS : Site | name : 'rtimer , op : free , AS >
       OE <- sr(signal, h, n) .
endm

mod ORC-SYSTEM is
  inc ORC-SITES .
  inc SYSTEM .
endm
```

A.3 Orc Real-Time and External Semantics

```
mod ORC-INIT-CONF is
  pr ORC-SYSTEM .
  inc SOCKET .

  var OE : EOid . var CF : Configuration .
  var d : Decl . var D : DeclList . var f : Expr .
  var PORT : Nat .

  --- ops to setup initial configuration
  op initCon : DeclList -> Env .
  op [_:_] : EOid Prog -> Configuration .
  op stdSites : -> Configuration .
  ops initClk initPubTrace : -> Configuration .

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

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

  eq stdSites = < let : Site | name : 'let ,    op : free , state : nil >
                < if : Site | name : 'if ,     op : free , state : nil >
                < rtimer : Site | name : 'rtimer , op : free , state : nil >
                < clock : Site | name : 'clock , op : free , state : nil > .
  eq initClk = < C1 : Clock | clk : c(0) > .

  eq initPubTrace = < T : PubTrace | t : nil > .

  --- create a configuration with standard sites (for expression objects)
  op E[_,_] : Nat Configuration -> LocalSystem .
  eq E[ PORT, CF ] = [<> CF stdSites initClk
               createServerTcpSocket(socketManager, C1, PORT, 10)] .

  --- variation without the clock server socket (for expression objects)
  op E[_]* : Configuration -> LocalSystem .
  eq E[ CF ]* = [<> CF stdSites initClk
               ] .


  --- create a configuration without standard sites (for site objects)
  op S[_,_] : Nat Configuration -> LocalSystem .
  eq S[ PORT, CF ] = [<> CF initClk
              createServerTcpSocket(socketManager, C1, PORT, 10)] .
endm

mod ORC-RWTIME is
  inc ORC-INIT-CONF .

  var OE : EOid . var OS : SOid . vars O O' OT OC LISTENER CLIENT : Oid .
  vars AS AS' : AttributeSet .
  vars CF CF' : Configuration .
  var h : Handle .
  var M : Qid . vars c c' : Const .
  var C : ConstList .   vars n d : Nat .
  var o : Object . var m : Msg .
  var CAttr : ClockAttr .
  var R : PubTrace .
  vars S S' IP : String .


  op delta : Configuration -> Configuration [frozen] .

  rl [InitClockSocket1] :
     < OC : Clock | clk : c(n) , AS > createdSocket(OC, socketManager, LISTENER)
     => < OC : Clock | clk : c(n) , AS > acceptClient(LISTENER, OC)
          [print "Clock server socket creatd. Awaiting connection from ticker ..."]
        .

  rl [InitClockSocket2] :
     < OC : Clock | clk : c(n) , AS > acceptedClient(OC, LISTENER, IP, CLIENT)
     => < OC : Clock | clk : c(n) , AS > receive(CLIENT, OC)
           [print "Ticker connected."]
        .

 crl [IncomingTick] :
     [ CF < OC : Clock | clk : c(n) , AS >
     received(OC, CLIENT, S) ]
     => [ delta(CF) < OC : Clock | clk : c(s(n)) , AS >
        receive(CLIENT, OC) ]
        if find(S, "#", 0) =/= notFound
           [print "Tick!"]
        .

 crl [WaitForTick] :
     < OC : Clock | clk : c(n) , AS >
     received(OC, CLIENT, S)
     => < OC : Clock | clk : c(n) , AS >
        receive(CLIENT, OC)
        if find(S, "#", 0) == notFound
     .

  rl [Halt] :
     < OC : Clock | clk : halt , AS > receive(CLIENT, OC)
     => < OC : Clock | clk : halt , AS > closeSocket(CLIENT, OC)
     ----     [print "Closing socket (@ receive)"]
     .

  rl [Halt] :
     < OC : Clock | clk : halt , AS > received(OC, CLIENT, S)
     => < OC : Clock | clk : halt , AS > closeSocket(CLIENT, OC)
     ----     [print "Closing socket (@ received)"]
     .

  rl [SocketClosed] :
     < OC : Clock | AS > closedSocket(OC, CLIENT, S)
     => < OC : Clock | AS >
         [print "Clock socket closed."] .

  eq delta(OS <- sc(OE, C, h, s(d)) CF) = OS <- sc(OE, C, h, d) delta(CF) .
  eq delta(O <- sr(c, h, s(d))      CF) = O <- sr(c, h, d)      delta(CF) .
  eq delta(O <- sr(c, O', s(d))     CF) = O <- sr(c, O', d)     delta(CF) .

  eq delta(CF) = CF [owise] .
endm

mod CONST-STRING-CONVERSION is
  inc PARAMETER .
  pr META-LEVEL .

  op error  : String    -> Const [ctor] .

  var C : ConstList .
  var  Q : Qid .
  var  QL : QidList .
  vars S S' S'' : String .
  var  N : Nat .

  op qidListString : QidList -> String .
  op qidListString : QidList String -> String .
  op stringQidList : String -> QidList .
  op stringQidList : String QidList -> QidList .

  eq qidListString(QL) = qidListString(QL, "") .
  eq qidListString(nil, S) = S .
  eq qidListString(Q QL, S) = qidListString(QL, S + string(Q) + " ") .

  eq stringQidList(S) = stringQidList(S, nil) .
  eq stringQidList("", QL) = QL .
  eq stringQidList(S, QL) = QL qid(S) [owise] .
  ceq stringQidList(S, QL)
    = stringQidList(S'', QL qid(S') )
    if N := find(S, " ", 0)
       /\ S' := substr(S, 0, N)
       /\ S'' := substr(S, N + 1, length(S)) .


  --- Const to String conversion
  op toString  : ConstList ~> String .
  eq toString(c(S)) = S .
  eq toString(C) = qidListString(metaPrettyPrint(upModule('PARAMETER, false), upTerm(C), none)) [owise] .

  --- String to Const conversion
  op toConst : String -> Const .
 ceq toConst(S)
   = downTerm(getTerm(metaParse(upModule('PARAMETER, false), stringQidList(S), 'Const)), error(S))
     if substr(S, 0, 6) == "signal"   ---- signal constant
     or substr(S, 0, 4) == "b `("     ---- bool
     or substr(S, 0, 4) == "i `("     ---- nat
     or substr(S, 0, 4) == "l `("     ---- list
     or substr(S, 0, 4) == "S `("     ---- Soid
      or substr(S, 0, 3) == "let"      ---- let site id
      or substr(S, 0, 2) == "if"       ---- if site id
      or substr(S, 0, 6) == "rtimer"   ---- rtimer site id
      or substr(S, 0, 5) == "clock"    ---- clock site id
      or substr(S, 0, 1) == "\""       ---- encoded singleton string
      or substr(S, 0, 7) == "_‘,_ ‘(" .  ---- ConstList
  eq toConst(S) = c(S) [owise] .       ---- arbitrary string

endm

mod ORC-EXT-OBJECTS is
  inc ORC-RWTIME .
  inc CONST-STRING-CONVERSION .
  pr ORC-PROXY .

  vars SR IP : String . var R : PubTrace .
  var OE : EOid . var xOS : XSOid .
  vars O LISTENER CLIENT OP OD OT : Oid . var c : Const .
  var C : ConstList . vars n PT : Nat . var M : Qid .
  var h : Handle . vars S S’ S’’ : String .
  var AS : AttributeSet . var ST : OState .
---  var PL : PubList .

  op sep : -> String .
  eq sep = "&" .

  ************************************
  **** Expressions as clients ****

  --- initiate an external site call
  eq [ExtSiteCall] :
    S(l(SR, PT), n) <- sc(OE, C, h, 0)
      = < P(OE,h) : Proxy | param : C, response : "" >
         createClientTcpSocket(socketManager, P(OE,h), SR, PT)
---          [print "Creating socket to " SR ":" PT " " n]
         .

  --- send the message
  rl [SendExtMessage] :
    createdSocket(OP, socketManager, OD)
    < OP : Proxy | param : C, response : S >
      => < OP : Proxy | param : C, response : S >
         send(OD, OP, (toString(C) + sep))
          [print "Sending " C " to " OD]
         .

  --- accumulate the response
  rl [RecExtMessage] :
    sent(OP, OD)
    < OP : Proxy | param : C, response : S >
      => < OP : Proxy | param : C, response : S >
         receive(OD, OP)
         ---- [print "Sent " C " to " OD]
         .

  rl [RecExtMessageCont] :
     received(OP, OD, S)
    < OP : Proxy | param : C, response : S’ >
      => < OP : Proxy | param : C, response : S’ + S >
         receive(OD, OP)
          [print "Response received: " S ]
         .

  --- process the response
  rl [ProcessExtMessage] :
    closedSocket(P(OE,h), OD, S)
    < P(OE,h) : Proxy | param : C, response : S’ >
      => OE <- sr(toConst(S’), h, 0)
```

```
        ----- [print "Socket closed to " OD]
        .

  **************************
  **** Sites as servers  ****

  --- create server socket when first started
  eq [InitializeSite] :
     < S(l(SR, PT), n) : Site | op : free, status : idle, AS >
     = < S(l(SR, PT), n) : Site | op : free, status : initializing, AS >
        createServerTcpSocket(socketManager, S(l(SR, PT), n), PT, 10)
          [print "Site " SR ":" PT " " n " initializing"]
        .

  rl [CreatedServerSocket] :
      < xOS : Site | op : free, status : initializing , AS >
      createdSocket(xOS, socketManager, LISTENER)
      => < xOS : Site | op : free, status : active , AS > acceptClient(LISTENER, xOS)
          [print "Site " xOS " ready"]
         .

  rl [AcceptedClient] :
      < xOS : Site | op : free, status : active, AS >
      acceptedClient(xOS, LISTENER, IP, CLIENT)
      => < xOS : Site | op : free, status : active , AS > receive(CLIENT, xOS)
         acceptClient(LISTENER, xOS)
          [print "Connected to " CLIENT ". Ready to receive"]
         .

 crl [AccumulateRequest] :
      < xOS : Site | op : free , buffer : S, AS > received(xOS, CLIENT, S')
      => < xOS : Site | op : free , buffer : S + S', AS >
         receive(CLIENT, xOS)
         if find(S + S', sep, 0) == notFound
         ---- [print "Appending response " S' " to " S " from " CLIENT]
         .

 crl [PrepareReply] :
      < xOS : Site | op : free , buffer : S, AS > received(xOS, CLIENT, S')
      => < xOS : Site | op : exec(toConst(substr(S'', 0, length(S'') + (- length(sep)))), CLIENT),
                    buffer : "", AS >
         if S'' := S + S'
         /\ n   := find(S'', sep, 0)
          [print "Received" S'' " from " CLIENT]
         .

  eq CLIENT <- sr(c, xOS, 0) < xOS : Site | AS > = < xOS : Site | AS > send(CLIENT, xOS, toString(c)) .

  rl [ReplySent] :
      < xOS : Site | AS > sent(xOS, CLIENT)
      => < xOS : Site | AS >
         closeSocket(CLIENT, xOS)
         ---- [print "Sent. Closing socket " CLIENT]
         .

  rl [ClosedClientSocket] :
      < xOS : Site | AS > closedSocket(xOS, CLIENT, S)
      => < xOS : Site | AS >
          [print "Socket closed: " CLIENT]
         .
endm

load socket

mod ORC-INTERFACE is
  inc ORC-EXT-OBJECTS .
endm
```

B Formal Model of Dist-Orc in Real-Time Maude

B.1 Orc Syntax and Explicit Substitution

The syntax and substitution modules in the formal model in Real-Time Maude are almost identical to those in the implementation (see Appendix A.1), and are thus omitted.

B.2 Orc Rewriting Semantics

The object-based, rewriting semantics specification in the formal model is essentially identical to the that in the impelmentation (Appendix A.2), but is given in an object-oriented specification style supported by Real-Time Maude, and is therefore omitted.

B.3 Socket Specification

```
(tomod DELAYS is

  class Delays |              *** A global (top-level) object abstracting communication delays
     ds : DelaySet .          *** A set of possible delays

  op did : -> Oid [ctor] .

  sort DelaySet .
  subsort Time < DelaySet .
  op mtd : -> DelaySet [ctor] .
  op __ : DelaySet DelaySet -> DelaySet [ctor assoc comm id: mtd] .

  var R : Time . var DS : DelaySet .

  eq R R DS = R DS .
endtom)

(tomod SOCKET is

  inc STRING .
  inc ORC-SYSTEM .
  pr ORC-PROXY .
  inc DELAYS .

  op socket : Nat -> Oid [ctor] .

  op createClientTcpSocket : Oid Oid String Nat -> Msg [ctor msg format (b o)] .
  op createServerTcpSocket : Oid Oid Nat Nat -> Msg [ctor msg format (b o)] .
  op createdSocket : Oid Oid Oid -> Msg [ctor msg format (m o)] .

  op acceptClient : Oid Oid -> Msg [ctor msg format (b o)] .
  op acceptedClient : Oid Oid String Oid -> Msg [ctor msg format (m o)] .

  op send : Oid Oid ConstList -> Msg [ctor msg format (b o)] .
  op sent : Oid Oid -> Msg [ctor msg format (m o)] .

  op receive : Oid Oid -> Msg [ctor msg format (b o)] .
  op received : Oid Oid ConstList Time -> Msg [ctor msg format (m o)] .

  op closeSocket : Oid Oid -> Msg [ctor msg format (b o)] .
  op closedSocket : Oid Oid String -> Msg [ctor msg format (m o)] .

  op socketError : Oid Oid String -> Msg [ctor msg format (r o)] .
  op socketManager : -> Oid .

--------------------------------------------------------------------------------
----
  class Manager | counter : Nat .

--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
----
  class Socket | endpoints : ProcessPair .

  sort ProcessPair .
  op `[_:_`] : Oid Oid -> ProcessPair [ctor comm] .

--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
----
  class ServerSocket | address : String, port : Nat, backlog : Nat .

  op server : Nat -> Oid [ctor] . ---- server socket objects identifiers

--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
----
  class Process | sys : LocalSystem .

  op pid : Nat -> Oid [ctor] . ---- process objects identifiers

--------------------------------------------------------------------------------

  vars SOCKET PID PID' O O' DID : Oid .
  vars OE : EOid . var iF : IExpr .
  vars DATA ADDRESS S S' : String .
  vars C C' : ConstList . var h : Handle .
  vars N M PORT BACKLOG : Nat .
  vars CONF CONF' : Configuration .
  var  MSG : Msg .
  vars DS DS' DS'' : DelaySet . var R : Time .

  rl [CreateServerTcpSocket] : < PID : Process |
        sys : [createServerTcpSocket(socketManager, O, PORT, BACKLOG) CONF] >
    < socketManager : Manager | counter : N >
    => < PID : Process |
          sys : [createdSocket(O, socketManager, server(N)) CONF] >
       < socketManager : Manager | counter : (N + 1) >
       < server(N) : ServerSocket |
          address : "localhost", port : PORT, backlog : BACKLOG > .

  ---- creating a client socket: success case
  rl [CreateClientSocketSuccess] :
    < PID : Process | sys : [acceptClient(server(N), O) CONF] >
    < PID' : Process |
        sys : [createClientTcpSocket(socketManager, P(OE,h), ADDRESS, PORT)
               < OE : Expr | exp : iF >
               CONF' ] >
    < socketManager : Manager | counter : M >
    < server(N) : ServerSocket | address : ADDRESS, port : PORT >
    => < PID : Process |
          sys : [acceptedClient(O, server(N), ADDRESS, socket(M)) CONF] >
       < PID' : Process |
          sys : [createdSocket(P(OE,h), socketManager, socket(M))
                 < OE : Expr | exp : iF >
```

```
                  CONF'] >
       < socketManager : Manager | counter : (M + 1) >
       < server(N) : ServerSocket | address : ADDRESS, port : PORT >
       < socket(M) : Socket | endpoints : [PID : PID'] > .

  ---- creating a client socket: fail case
  rl [CreateClientSocketFail] : < PID : Process | sys : [acceptClient(server(N), O) CONF] >
    < PID' : Process |
        sys : [createClientTcpSocket(socketManager, P(OE,h), ADDRESS, PORT)
               < OE : Expr | exp : iF >
               CONF'] >
    < socketManager : Manager | counter : M >
    < server(N) : ServerSocket | address : ADDRESS, port : PORT >
    => < PID : Process | sys : [acceptClient(server(N), O) CONF] >
    < PID' : Process |
        sys : [socketError(P(OE,h), socketManager, "")
               < OE : Expr | exp : iF >
               CONF'] >
    < socketManager : Manager | counter : M >
    < server(N) : ServerSocket | address : ADDRESS, port : PORT >  .

    ---- send/receive
    ---- The sender is an Expr Object
 crl [send1] : < PID : Process | sys : [send(SOCKET, P(OE,h), C)
                                       < OE : Expr | exp : iF > CONF] >
    < PID' : Process | sys : [receive(SOCKET, O') CONF'] >
    < SOCKET : Socket | endpoints : [PID : PID'] >
    < DID : Delays | ds : DS >
    => < PID : Process | sys : [sent(P(OE,h), SOCKET)
                                 < OE : Expr | exp : iF > CONF] >
       < PID' : Process | sys : [received(O', SOCKET, C, R) CONF'] >
       < SOCKET : Socket | endpoints : [PID : PID'] >
       < DID : Delays | ds : DS >
      if DS' R DS'' := DS .

    ---- The receiver is an Expr Object
 crl [send2] : < PID : Process | sys : [send(SOCKET, O, C) CONF] >
    < PID' : Process | sys : [receive(SOCKET, P(OE,h))
                               < OE : Expr | exp : iF > CONF'] >
    < SOCKET : Socket | endpoints : [PID : PID'] >
    < DID : Delays | ds : DS >
    => < PID : Process | sys : [sent(O, SOCKET) CONF] >
       < PID' : Process | sys : [received(P(OE,h), SOCKET, C, R)
                                 < OE : Expr | exp : iF > CONF'] >
       < SOCKET : Socket | endpoints : [PID : PID'] >
       < DID : Delays | ds : DS >
      if DS' R DS'' := DS .

    ---- close message
  eq [close] :
    < PID : Process | sys : [closeSocket(SOCKET, O) CONF] >
    < PID' : Process | sys : [receive(SOCKET, O') CONF'] >
    < SOCKET : Socket | endpoints : [PID : PID'] >
    = < PID : Process | sys : [closedSocket(O, socketManager, "") CONF] >
       < PID' : Process | sys : [closedSocket(O', socketManager, "") CONF'] >
       .
endtom)
```

B.4 Orc Real-Time and External Semantics

```
(tomod ORC-PRIORITY is
  inc ORC-SYSTEM .
  inc SOCKET .
  op eagerEnabled : LocalSystem -> [Bool] [frozen] .
  op eagerEnabled : GlobalSystem -> [Bool] [frozen] .

  var OE : EOid .  var OS : SOid . var xOS : XSOid .
  vars PID PID' DID SOCKET O O' O'' O1 O2 : Oid .
  vars AS Atts Atts' Atts'' Atts''' : AttributeSet .
  vars ADDRESS DATA S : String . vars PORT N M i : Nat .
  vars CF CF' CONF CONF' : Configuration . var h : Handle . var c : Const .
  var x : Var . var P : AParamList . var C : ConstList .
  vars f f' : Expr . var aF : AExpr . var iF : IExpr . var E : ExprName .
  var DS : DelaySet .

  --- eager global configurations
 eq eagerEnabled({ < PID : Process | sys : [< OE : Expr | exp : aF > CF] > CF' } ) =  true .
ceq eagerEnabled({ < PID : Process | sys :
           [OE <- sr(c, h, 0) < OE : Expr | exp : iF > CF] > CF' } ) =  true
      if h in handles(iF) .

  eq eagerEnabled({
        < PID : Process | sys : [acceptClient(server(N), O) CONF] >
        < PID' : Process | sys :
                   [createClientTcpSocket(socketManager, O', ADDRESS, PORT) CONF'] >
        < socketManager : Manager | counter : M >
        < server(N) : ServerSocket | address : ADDRESS, port : PORT > CF }) = true .

  eq eagerEnabled({
        < PID : Process | sys : [send(SOCKET, O, C) CONF] >
        < PID' : Process | sys : [receive(SOCKET, O') CONF'] >
        < SOCKET : Socket | endpoints : [PID : PID'] >
        < DID : Delays | ds : DS > CF}) = true .

  eq eagerEnabled({
        < PID : Process | sys : [createServerTcpSocket(socketManager, O, PORT, N) CONF] >
        < socketManager : Manager | counter : M > CF}) = true .

endtom)


(tomod ORC-INIT-CONF is
  inc ORC-PRIORITY .

  var OE : EOid . var CF : Configuration .
  var d : Decl . var D : DeclList . var f : Expr .
  var PORT : Nat .

  --- ops to setup initial configuration
  op initCon : DeclList -> Env .
  op `[_:_`] : EOid Prog -> Configuration .
  op stdSites : -> Configuration .
  ops initClk initPubTrace : -> Configuration .

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

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

  eq stdSites = < let : Site | name : 'let ,    op : free >
               < if : Site | name : 'if ,     op : free >
               < rtimer : Site | name : 'rtimer , op : free >
               < clock : Site | name : 'clock , op : free > .

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

  --- create a configuration with standard sites (for expression objects)
  *** PORT is only mentioned for compatibility with the real time semantics; it is
  ***   ignored in the logical semantics
  op E`[_`,_`] : Nat Configuration -> LocalSystem .
  eq E[ PORT, CF ] = [ CF stdSites initClk ] .

  --- variation without the clock server socket (for expression objects)
  op E`[_`,_`]* : Nat Configuration -> LocalSystem .
  eq E[ PORT, CF ]* = [ CF stdSites initClk ] .

  --- create a configuration without standard sites (for site objects)
  op S`[_`,_`] : Nat Configuration -> LocalSystem .
  eq S[ PORT, CF ] = [ CF initClk ] .

endtom)

(tomod ORC-RWTIME is
  inc ORC-INIT-CONF .

  var OE : EOid . var OS : SOid . vars O O' PID OT OC LISTENER CLIENT : Oid .
  var iF : IExpr .
  vars CF CF' Conf : Configuration .
  var h : Handle .
  var M : Qid . vars c c' : Const .
  var C : ConstList .   vars n t : Nat .
  var o : Object . var m : Msg .
  var CAttr : ClockAttr .
  vars S S' IP : String .
  vars R R' d : Time .


  crl [tick] : { < PID : Process | sys : [< OC : Clock | clk : c(R) > CF] > Conf }
               => { < PID : Process | sys : [< OC : Clock | clk : c(R plus R') >
                    delta(CF, R')] > delta(Conf, R') } in time R'
        if eagerEnabled( { < PID : Process | sys : [< OC : Clock | clk : c(R) > CF] > Conf } )
           =/= true
        /\ R' le mte(< PID : Process | sys : [< OC : Clock | clk : c(R) > CF] > Conf) [nonexec]
      .

  op delta : Configuration Time -> Configuration [frozen] .

  eq delta(< PID : Process | sys : [CF] > Conf, R)
     = < PID : Process | sys : [delta(CF, R)] > delta(Conf, R) .

  eq delta(< OC : Clock | clk : c(R) > CF, R')
     = < OC : Clock | clk : c(R plus R') > delta(CF, R') .
  eq delta(OS <- sc(OE, C, h, d) CF, R') = OS <- sc(OE, C, h, d monus R') delta(CF, R') .
  eq delta(O <- sr(c, h, d) CF, R') = O <- sr(c, h, d monus R') delta(CF, R') .
  eq delta(O <- sr(c, O', d) CF, R') = O <- sr(c, O', d monus R') delta(CF, R') .
  eq delta(received(O, O', C, d) CF, R') = received(O, O', C, d monus R') delta(CF, R') .

  eq delta(CF, R') = CF [owise] .

  op mte : Configuration -> TimeInf [frozen] .

  eq mte(< PID : Process | sys : [CF] > Conf) = minimum(mte(CF), mte(Conf)) .

  eq mte(OS <- sc(OE, C, h, d) CF) = minimum(d, mte(CF)) .
 ceq mte(< OE : Expr | exp : iF > OE <- sr(c, h, d) CF)
     = minimum(d, mte(< OE : Expr | exp : iF > CF)) if h in handles(iF) .
  eq mte(O <- sr(c, O', d) CF) = minimum(d, mte(CF)) .
  eq mte(received(O, O', C, d) CF) = minimum(d, mte(CF)) .

  eq mte(CF) = INF [owise] .
endtom)

(tomod ORC-EXT-OBJECTS is
  inc ORC-RWTIME .
  pr ORC-PROXY .

  vars SR IP : String .
  var OE : EOid . var xOS : XSOid .
```

```
vars O LISTENER CLIENT OP OD OT : Oid . var c : Const .
var C : ConstList . vars n PT : Nat . var M : Qid .
var h : Handle . vars S S' S'' : String .
var AS : AttributeSet . var ST : OState .
var B : Bool .

**** Expressions as clients ****

--- initiate an external site call
eq [ExtSiteCall] :
  S(l(SR, PT), n) <- sc(OE, C, h, 0)
    = < P(OE,h) : Proxy | param : C, response : nilA >
       createClientTcpSocket(socketManager, P(OE,h), SR, PT) .

---- Do nothing on socketError .

--- send the message
eq [SendExtMessage] :
  createdSocket(OP, socketManager, OD)
  < OP : Proxy | param : C, response : nilA >
    =  < OP : Proxy | param : C, response : nilA >
       send(OD, OP, C) .

--- accumulate the response
eq [RecExtMessage] :
  sent(OP, OD)
  < OP : Proxy | param : C, response : nilA >
    =  < OP : Proxy | param : C, response : nilA >
       receive(OD, OP) .

eq [RecExtMessageCont] :
  received(OP, OD, c, 0)
  < OP : Proxy | param : C, response : nilA >
    =  < OP : Proxy | param : C, response : c >
       receive(OD, OP) .

--- process the response
eq [ProcessExtMessage] :
  closedSocket(P(OE,h), OD, S)
  < P(OE,h) : Proxy | param : C, response : c >
    =  OE <- sr(c, h, 0) .

**** Sites as servers  ****

--- create server socket when first started
eq [InitializeSite] :
   < S(l(SR, PT), n) : XSite | op : free, status : idle >
   = < S(l(SR, PT), n) : XSite | op : free, status : initializing >
      createServerTcpSocket(socketManager, S(l(SR, PT), n), PT, 10) .

---- Do nothing on socketError .

eq [CreatedServerSocket] :
    < xOS : XSite | op : free, status : initializing  >
    createdSocket(xOS, socketManager, LISTENER)
    =  < xOS : XSite | op : free, status : active  > acceptClient(LISTENER, xOS) .

eq [AcceptedClient] :
    < xOS : XSite | op : free, status : active > acceptedClient(xOS, LISTENER, IP, CLIENT)
    =  < xOS : XSite | op : free, status : active  > receive(CLIENT, xOS)
       acceptClient(LISTENER, xOS) .

eq [PrepareReply] :
    < xOS : XSite | op : free  > received(xOS, CLIENT, C, 0)
    =  < xOS : XSite | op : exec(C, CLIENT) > .


eq CLIENT <- sr(c, xOS, 0) < xOS : XSite | > = < xOS : XSite | > send(CLIENT, xOS, c) .
```

```
  eq [ReplySent] :
      < xOS : XSite | > sent(xOS, CLIENT)
      =  < xOS : XSite | >
         closeSocket(CLIENT, xOS) .

  eq [ClosedClientSocket] :
      < xOS : XSite | > closedSocket(xOS, CLIENT, S) =  < xOS : XSite | > .
endtom)


(tomod ORC-INTERFACE is
  inc ORC-EXT-OBJECTS .
endtom)
```

C The Dist-Auction Example

C.1 Dist-Auction in Maude

```
mod ITEMS-N-REQUESTS is
  inc ORC-INTERFACE .

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

  sorts Item ItemList .
  subsort Item < ItemList .

  op iNil : -> ItemList .
  op _,_ : ItemList ItemList -> ItemList [assoc id: iNil] .

  op itms : ItemList -> OState .

  sorts Req ReqList .
  subsort Req < ReqList .

  op rNil : -> ReqList .
  op _,_ : ReqList ReqList -> ReqList [assoc id: rNil] .

  op reqs : ReqList -> OState .

  op item : Nat Nat Nat -> Item .
  op req : Oid -> Req .
endm

mod SELLER-SITE is
  inc ITEMS-N-REQUESTS .

  vars O OS : Oid . var AS : AttributeSet .
  var IT : ItemList . vars i id t m : Nat .

  eq < OS : Site | name : 'seller , op : exec("postNext", O) ,
                   state : itms(item(id, t, m), IT) , AS >
     = < OS : Site | name : 'seller , op : free , state : itms(IT)  , AS >
       O <- sr(l(i(id), i(t), i(m)), OS, O)
          [print "Publishing next item " id]
       .

  op sellerSid : Nat -> XSOid .
  eq sellerSid(i) = S(l("localhost", 44800 + i), i) .

  op sellerSite : Nat -> Object .
  eq sellerSite(i) = < sellerSid(i) : Site | name : 'seller, op : free ,
                       state : itms(item(1910, 5, 500), item(1720, 7, 700)) ,
                      status : idle , buffer : "" > .
  op initSeller : Nat -> LocalSystem .
  eq initSeller(i) = S[ 54800 + i, sellerSite(i) ] .
endm


mod AUCTION-SITE is
  inc ITEMS-N-REQUESTS .

  vars O O' OS : Oid . var AS : AttributeSet .
  var IT : ItemList . vars id t m n k : Nat .
  var RQ : ReqList . var WN : WItemSet .
  var OST : OState .

  op item : Nat Nat Nat Oid -> Item .


  sorts WItem WItemSet .
  subsort WItem < WItemSet .

  op winner : Nat Nat Nat -> WItem .
  op noWinners : -> WItemSet .
  op _,_ : WItemSet WItemSet -> WItemSet [assoc comm id: noWinners] .

  op won : WItemSet -> OState .

  --- post message
  eq < OS : Site | name : 'auction , op : exec(("post", l(i(id), i(t), i(m))), O) ,
                 state : itms(IT) OST , AS >
     = < OS : Site | name : 'auction , op : free ,
                   state :  itms(IT, item(id, t, m, O)) OST, AS >
         [print "Item " id " posted"]
       .

  --- get message
  eq < OS : Site | name : 'auction , op : exec("getNext", O) , state : reqs(RQ) OST , AS >
     = < OS : Site | name : 'auction , op : free , state : reqs(RQ, req(O)) OST , AS > .


  --- servicing a request
  eq < OS : Site | name : 'auction , op : free ,
                  state : reqs(req(O) , RQ) itms(item(id, t, m, O'), IT) OST , AS >
     = < OS : Site | name : 'auction , op : free ,
                   state : reqs(RQ) itms(item(id, t, m, O'), IT) OST , AS >
       ----O' <- sr(signal, OS, O)
       O <- sr(l(i(id), i(t), i(m)), OS, O)
         [print "Bidding to start for item " id]
       .

  --- won message
  eq < OS : Site | name : 'auction , op : exec(("won", i(n), i(id), i(m)), O) ,
                 state : won(WN) itms(item(id, t, k, O'), IT) OST, AS >
     = < OS : Site | name : 'auction , op : free ,
                   state : won(winner(n, id, m), WN) itms(IT) OST , AS >
       O <- sr(signal, OS, O)
       O' <- sr(signal, OS, O)
         [print "Item " id " won by Bidder " n "!"]
       .

  op auctionSid : -> XSOid .
  eq auctionSid = S(l("localhost", 44600), 0) .

  op auctionSite : -> Object .
  eq auctionSite = < auctionSid : Site | name : 'auction , op : free ,
                   state : reqs(rNil) itms(iNil) won(noWinners), status : idle , buffer : "" > .

  op initAuction : -> LocalSystem .
  eq initAuction = S[ 54600, auctionSite ] .
endm
```

```
mod BIDDERS-SITE is
  inc ITEMS-N-REQUESTS .

  vars O OS : Oid . var AS : AttributeSet .
  vars OL OL' : OState . vars id i m RT n b : Nat .
  var IP : String . var IBS : ItemBidSet . var C : ConstList .
  vars BS BS' : BidderSet .

  sorts Bidder BidderSet .
  subsort Bidder < BidderSet .

  op b : Nat ItemBidSet -> Bidder .
  op noBidders : -> BidderSet .
  op __ : BidderSet BidderSet -> BidderSet [assoc comm id: noBidders] .

  sorts ItemBid ItemBidSet .
  subsorts ItemBid < ItemBidSet .

  op [_,_] : Nat Nat -> ItemBid .
  op noBids : -> ItemBidSet .
  op __ : ItemBidSet ItemBidSet -> ItemBidSet [assoc comm id: noBids] .

  op bidders : BidderSet -> OState .

  --- bidList message
 ceq < OS : Site | name : 'bidders , op : exec(("nextBidList", i(id), i(m)), O) ,
                  state : bidders(BS) , AS >
    = < OS : Site | name : 'bidders , op : free ,
                  state : bidders(BS') , AS >
      O <- sr(l(C), OS, 0)
   if {C, BS'} := nextBL(id, m, BS)
         [print "Next bid list published for item " id]
     .

  sort SPair .
  op {_,_} : Const BidderSet -> SPair .

  op nextBL : Nat Nat BidderSet -> SPair .
  eq nextBL(id, m, BS) = nextBL*(id, m, BS, nilA, noBidders) .

  op nextBL* : Nat Nat BidderSet ConstList BidderSet -> SPair .
  eq nextBL*(id, m, noBidders, C, BS') = {C, BS'} .
  eq nextBL*(id, m, (b(n, [id, b] IBS) BS), C, BS')
     = nextBL*(id, m, BS, (C, l(i(m + n * 10),i(n))), BS' b(n, [id, m + n * 10] IBS)) .
  eq nextBL*(id, m, (b(n, IBS) BS), C, BS')
     = nextBL*(id, m, BS, (C, l(i(m + n * 10),i(n))), BS' b(n, [id, m + n * 10] IBS)) [owise] .

  op biddersSid : -> XSOid .
  eq biddersSid = S(l("localhost", 44400), 0) .

  op biddersSite : -> Object .
  eq biddersSite = < biddersSid : Site | name : 'bidders, op : free ,
                       state : bidders(b(1, noBids) b(2, noBids) b(3, noBids) ) ,
                      status : idle , buffer : "" > .

  op initBidders : -> LocalSystem .
  eq initBidders = S[ 54400, biddersSite ] .

endm

mod MAXBID-SITE is
  inc ORC-INTERFACE .

  op mb : Const ConstList -> Const .

  op non : -> SOid .
  vars O O' OS : Oid . var AS : AttributeSet .
  var c : Const . var C : ConstList . vars n n' id id' : Nat .

  ---  computing the largest bid
  eq < OS : Site | name : 'maxBid , op : exec(l(c, C), O) , state : nil , AS >
     = < OS : Site | name : 'maxBid , op : free , state : nil , AS >
       O <- sr(mb(l(i(0), i(0)), (c, C)), OS, 0)
       .

  eq mb(l(i(n), i(id)), (l(i(n'), i(id')) , c, C)) =
       if (n' > n) then mb(l(i(n'), i(id')), (c, C))
       else           mb(l(i(n), i(id)), (c, C))  fi .
  eq mb(l(i(n), i(id)), l(i(n'), i(id'))) =
       if (n' > n) then l(i(n'), i(id'))
       else           l(i(n), i(id)) fi .

  op maxBidSid : -> XSOid .
  eq maxBidSid =  S(l("localhost", 44700), 0) .

  op maxBidSite : -> Object .
  eq maxBidSite = < maxBidSid : Site | name : 'maxBid , op : free ,
                  state : nil , status : idle , buffer : "" > .

  op initMaxBid : -> LocalSystem .
  eq initMaxBid = S[ 54700, maxBidSite ] .
endm

mod AUCTION-SITES is
  inc SELLER-SITE .
  inc AUCTION-SITE .
  inc BIDDERS-SITE .
  inc MAXBID-SITE .
endm

mod AUCTION-MANAGER is
  inc AUCTION-SITES .

  vars i n : Nat .

  ops Posting Bidding Bids Collect TimeoutRound BiddingRound : -> ExprName .
  ops PostingDecl BiddingDecl BidsDecl CollectDecl TimeoutRoundDecl BiddingRoundDecl : -> Decl .
  ops PostingExpr BiddingExpr : -> Expr .
  ops seller x d h t t1 t2 bl m winner name id
      bid bidlist minbid maxbid mbid mbidder nd xs : -> Var .


  eq PostingDecl =
     Posting seller :=
        seller{0}("postNext") > x > auctionSid("post", x{0}) >> rtimer(i(1)) >>
        Posting(seller{0}) .

  eq BiddingDecl =
     Bidding :=
        auctionSid("getNext") > x >
        it(i(0), x{0}) > id >
        it(i(1), x{0}) > d >
        it(i(2), x{0}) > m >
        Bids(id{0}, d{0}, m{0}, i(0)) > winner >
        it(i(0), winner{0}) > bid > it(i(1), winner{0}) > name >
        ( if(eq(name{0}, i(0))) >> Bidding()
        | if(neg(eq(name{0}, i(0)))) >> auctionSid("won", name{0}, id{0}, bid{0}) >> Bidding()
        ) .

  eq BidsDecl =
     Bids (id, d, bid, winner) :=
        ( if(gte(i(0), d{0})) >> let(bid{0}, winner{0})
        | if(neg(gte(i(0), d{0}))) >> clock() > t1 > min(d{0}, i(1)) > t >
```

```
            TimeoutRound(id{0}, bid{0}, t{0}) > maxbid >
            ( if(eq(maxbid{0}, signal)) >> sub(d{0},t{0}) > nd >
                   Bids(id{0}, nd{0}, bid{0}, winner{0})
            | if(neg(eq(maxbid{0}, signal))) >>
                head(maxbid{0}) > mbid > it(i(1), maxbid{0}) > mbidder >
                clock() > t2 > sub(t2{0},t1{0}) > x >
                rtimer(i(1)) >> sub(d{0},x{0}) > nd > sub(nd{0}, i(1)) > nd >
                  Bids(id{0}, nd{0}, mbid{0}, mbidder{0})
            )
        ) .

  eq TimeoutRoundDecl =
     TimeoutRound (id, bid, t) :=
         let(x{0}) < x < ( rtimer(t{0}) | BiddingRound(id{0}, bid{0}) ) .

  eq BiddingRoundDecl =
     BiddingRound (id, bid) := biddersSid("nextBidList", id{0}, bid{0}) > bidlist >
          maxBidSid(bidlist{0}) .

  eq PostingExpr = Posting(sellerSid(0)) .
  eq BiddingExpr = Bidding() .

  op postingEid : -> EOid .
  eq postingEid = E(l("localhost", 44200), 0) .
  op initPosting : -> LocalSystem .
  eq initPosting = E[ 54200, [ postingEid : PostingDecl ; rtimer(i(1)) >> PostingExpr ] ] .

  op biddingEid : -> EOid .
  eq biddingEid = E(l("localhost", 44300), 0) .
  op initBidding : -> LocalSystem .
  eq initBidding = E[ 54300, [ biddingEid : BiddingDecl ; BidsDecl ; TimeoutRoundDecl ;
                               BiddingRoundDecl ; rtimer(i(1)) >> BiddingExpr ] ] .

endm

set print attr on .
```

C.2 Formal Analysis of Dist-Auction in Real-Time Maude

The specification of the DIST-AUCTION example itself is identical to that in Section C.1. We give below the module defining the atomic predicates and the LTL formulas.

```
in auction-manager.maude

(tomod AUCTION-PREDS is
  pr AUCTION-MANAGER .
  inc TIMED-MODEL-CHECKER .

  vars PID PID' O O' : Oid .
  vars AS AS' Atts Atts' : AttributeSet .
  vars n n' m m' id : Nat .
  vars Conf Conf' CF CF' : Configuration .
  vars OL OL' OL1 OL1' OL2 OL2' OST : OState .
  vars WN WN' : WItemSet . var IBS : ItemBidSet .
  vars BS BS' : BidderSet .
  var S : String . var R : Time . var DS : DelaySet .

  op commError : -> Prop .

  eq { < PID : Process | sys : [socketError(O, O', S) CF] > Conf } |= commError = true .

  ops hasBid sold conflict : Nat -> Prop .

  eq { < PID : Process | sys : [< O : XSite | name : 'bidders, state :
          bidders(b(n, [id, m] IBS) BS) OL > CF] > Conf } |= hasBid(id) = true .

  eq { < PID : Process | sys : [< O : XSite | name : 'auction, state :
          won(winner(n, id, m), WN) OST > CF] > Conf } |= sold(id) = true .

  eq { < PID : Process | sys : [< O : XSite | name : 'auction , state :
          won(winner(n, id, m), winner(n', id, m'), WN) OST > CF] > Conf } |= conflict(id) = true .

  op maxbid : Nat BidderSet Nat -> Nat [frozen] .
  eq maxbid(id, b(n, [id, m'] IBS) BS, m) = maxbid(id, BS, max(m,m')) .
  eq maxbid(id, noBidders, m) = m [owise] .

  op max : Nat -> Prop .
 ceq { < PID : Process | sys : [< O : XSite | name : 'auction, state :
                         won(winner(n, id, m), WN) OST > CF] >
      < PID' : Process | sys : [< O' : XSite | name : 'bidders, state :
                         bidders(BS) OL > CF'] > Conf } |= max(id)
    = true if m == maxbid(id, BS, m) .

  op boundedDelay : Time -> Prop .

 ceq { < O : Delays | ds : DS > Conf } |= boundedDelay(R) = true if R >= maxDelay(DS) .

  op maxDelay : DelaySet -> Time .
  eq maxDelay(R DS) = max(R, maxDelay(DS)) .
  eq maxDelay(mtd) = 0 .
endtom)

(tomod AUCTION-MODEL-CHECK is
  pr AUCTION-PREDS .

  var id : Nat . var DS : DelaySet . var R : Time .

  op commit : Nat -> Formula .
  eq commit(id) = hasBid(id) -> <> sold(id) .

  op commitAll : -> Formula .
  eq commitAll = [] (commit(1910) /\ commit(1720)) .

  op commitAllNoErrors : -> Formula .
  eq commitAllNoErrors = ([] ~ commError) -> [] (commit(1910) /\ commit(1720)) .

  op winmax : Nat -> Formula .
  eq winmax(id) = sold(id) -> max(id) .

  op winmaxAll : -> Formula .
  eq winmaxAll = [] (winmax(1910) /\ winmax(1720)) .

  op winmaxAllNoErrors : -> Formula .
  eq winmaxAllNoErrors = ([] ~ commError) -> [] (winmax(1910) /\ winmax(1720)) .

  op uniqueWinner : Nat -> Formula .
  eq uniqueWinner(id) = ~ conflict(id) .

  op uniqueWinnerAll : -> Formula .
  eq uniqueWinnerAll = [] (uniqueWinner(1910) /\ uniqueWinner(1720)) .

  op initial : DelaySet -> GlobalSystem .
  eq initial(DS) = {
      < did : Delays | ds : DS >
      < socketManager : Manager | counter : 0 >
      < pid(0) : Process | sys : initSeller(0) >
      < pid(1) : Process | sys : initAuction >
      < pid(2) : Process | sys : initBidders >
      < pid(5) : Process | sys : initMaxBid >
      < pid(6) : Process | sys : initPosting >
      < pid(7) : Process | sys : initBidding > }
      .
  op dinitial : -> GlobalSystem .
  eq dinitial = initial(1/10 2/10) .
endtom)

---- maximal time sampling strategy
(set tick max .)
```