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

Musab AlTurki
The University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
alturki@cs.illinois.edu

José Meseguer
The University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
meseguer@cs.illinois.edu

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 by defining an abstract model of time and the socket communication infrastructure, 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 paper addresses all these issues for the Orc programming language [1], an elegant and powerful language for orchestration of web services. Orc has a well-developed theoretical basis [1, 2, 3, 4, 5] and also a well-engineered language implementation [6, 7]. 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 paper 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 [8], 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* [9, 10], which brought the Orc language definition closer to an actual distributed implementation while preserving semantic equivalence. The third step, taken in this paper, 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 [10], 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 [11], 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 paper, 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* contribution is the *correct-by-construction* nature of our distributed Orc implementation, henceforth referred to as DIST-ORC. This contribution builds on the semantic equivalences already proved in [9] between the SOS, reduction semantics, and object-based semantics levels. In this way, a seamless path from formal language definition to a correct distributed implementation is obtained.

Our *second* contribution is to show 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 [12].

A *third* 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 [11], but it is done here for

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

Figure 1: Syntax of Orc

real-time systems for the first time.

The paper is organized as follows. In Section 2, we give an overview of the Orc theory and its informal semantics, and describe in some detail the AUCTION example in Orc. In Section 3, we briefly review Orc's rewriting semantics previously developed that form the basis of the distributed implementation DIST-ORC, which is presented and discussed in Section 4, along with a distributed implementation `Dist-Auction` of the auction example. Section 5 describes the formal model of DIST-ORC in Real-Time Maude, with which formal analysis of distributed Orc applications, such as `Dist-Auction`, can be carried out. The paper is concluded with a summary and a brief 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. A site may also represent the interaction with a human being, most commonly within the context of business workflows [13]. 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 value d. When a site responds to a call with a 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.

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 exactly 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. Orc also assumes a few more internal sites, which are needed for effective programming. 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 $\bar{x}$, and (3) *clock*, which publishes the current time value.

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 only three combinators, which were shown to be capable of expressing a wide variety of distributed computations succinctly and elegantly [1]. 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. 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 ($M(\bar{p})$ and $E(\bar{p})$, respectively); or (3) the composition of two expressions by one of the three composition operators:

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

Sequential composition, $f > x > g$, 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, $f < x < g$, 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 message to a, depending on which site publishes a value first. If neither site publishes, the variable x is not bound to a concrete value, and the call to *Email* is not made.

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. For example, the Orc expression $\mathit{let}(x) < x < (M \mid \mathit{rtimer}(t) \gg \mathit{let}(0))$ specifies a timeout on the call to a site M. Upon executing it, 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 units, then either c or 0 is published. Otherwise, 0 is published. Another example is the two-branch conditional **if** b **then** f **else** g, which can be succinctly specified in Orc as the 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. These examples and many more can be found in [1].

The Auction 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 [1], 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 bids higher than the current bid 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.

Figure 2 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 and 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 then announces the winning bidder before proceeding to the next item. An item is a tuple (id,d,m), with id the item identifier, d its auction duration, and m the starting bid. We use integer subscripts in variables to pick elements of a tuple, e.g. x_0 is the first element of the tuple given by x, and so on.

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

Figure 2: Orc expressions in the AUCTION program

The declarations in Figure 2 along with the expression $\mathit{Posting}(s) \mid \mathit{Bidding}$, for a given seller site s, fully specify in Orc the *Auction* program:

$$\mathit{Posting}\ ;\ \mathit{Bidding}\ ;\ \mathit{Bids}\ ;\ \mathit{TimeoutRound}\ ;\ \mathit{Posting}(s) \mid \mathit{Bidding}$$

3 Rewriting Semantics of Orc

Rewriting logic [14] 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 [15, 16, 17] and references there). Furthermore, with the availability of high-performance rewriting logic implementations, such as Maude [11], language specifications can both be executed and model checked.

In previous work [9], we have developed an executable specification giving a formal semantics to Orc in rewriting logic. The specification was shown to capture Orc's intended *synchronous* semantics [10], 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 [8, 10, 9] in order to achieve greater efficiency and expressiveness.

The distributed implementation presented in this paper builds on the object-based semantics $\mathscr{R}_{Orc}$, which was first introduced in [9]. This semantics generalizes the previous semantics to multiple Orc expressions and makes explicit the interactions between Orc site and expression objects. The extension to object-based semantics enabled defining arbitrarily complex applications in Orc. We give a quick overview of the object-based semantics below.

In $\mathscr{R}_{Orc}$, the state of an Orc program is defined as a *configuration* (of sort $\mathsf{Configuration}$) of objects and messages. Configurations are multisets specified by the associative and commutative empty juxtaposition operator $\_\_ : \mathsf{Configuration}\ \mathsf{Configuration} \to \mathsf{Configuration}$, with the empty multiset *null* as the identity element. An object is a term of the form $\langle I : \mathscr{C} \mid AS \rangle$, with I a unique object identifier, $\mathscr{C}$ the class name of the object, and AS a set of attribute-value pairs, each of the form $\mathit{attr} : \mathit{value}$, where attr is the attribute's name, and value is the corresponding value. There are two main classes of Orc objects: *Expression* objects and *Site* objects, corresponding, respectively, to Orc expressions and sites. An Orc expression object for an Orc program $\bar{d}\ ;\ f$ maintains a set of expression declarations corresponding to

$\bar{d}$ in an attribute *env*, and the expression f to be evaluated in an attribute *exp*. An Orc site object has a *name* attribute for the site's name, and maintains the site's state in a *state* attribute.

In keeping with the philosophy of the Orc theory, expression objects are modeled as active objects with 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. 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 O, a site call expression $M(C)$ causes a site call message of the form $M \leftarrow sc(O,C,R)$ to be emitted into the configuration, with R a non-negative value representing the delay of the message; that is, the time it takes for the message to reach the site M. Once the message is received and processed by M, the site may reply back with a site return message $O \leftarrow sr(M,c,R')$, with c the value published by M, and R' a message transmission delay.

Timed behaviors are specified using a *tick* rule that advances logical time and manages the effect of time elapse by appropriately updating message delays. 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 [18], such that only a finite number of instantaneous transitions are possible within any finite period of time [9].

In 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* [17] and their implementations in Real-Time Maude [19], 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.

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, an 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 Orc rewriting semantics specification $\mathscr{R}_{Orc}$ outlined in Section 3.

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 underlying 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 [11]. We give a brief overview of Maude's implementation of sockets below. We then 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 is available online at `http://www.cs.illinois.edu/homes/alturki/dist-orc`.

4.1 Sockets and External Objects in Maude

Maude provides a low-level implementation of sockets, which effectively enables a Maude process to exchange messages with other processes, including other Maude instances, according to the connection-oriented TCP communication protocol. More specifically, a configuration `CF` that contains a *socket portal*, which is a predefined constant `<>` of sort `Portal` (which is a subsort of `Configuration`), may communicate with objects external to the Maude process executing `CF` through a set of special messages defining an interface to Maude sockets. Assuming that `O` is an identifier for an object in `CF`, these messages are as follows:

1. `createClientTcpSocket(socketManager, O, ADDRESS, PORT)`, which asks Maude's socket manager, a factory for socket objects, for a client socket to a server located at `ADDRESS:PORT`. Maude then responds with either a `createdSocket(O, socketManager, SOCKET)` message, indicating successful creation of the client socket `SOCKET`, or a `socketError(O, socketManager, S)` message, with `S` a string briefly describing the reason for failure.
2. `send(SOCKET, O, S)`, which asks for the string `S` to be sent through `SOCKET`. This message elicits either a message `sent(O, SOCKET)`, when the string is successfully sent, or a message `closedSocket (O, SOCKET, S)`, if an error occurred.
3. `receive(SOCKET, O)`, which solicits a response through `SOCKET`. When a response is received, Maude issues the message `received(O, SOCKET, S)`, with `S` the string received. In case of an error, the socket is closed with the message `closedSocket(O, SOCKET, S)`.
4. `createServerTcpSocket(socketManager, O, PORT, BACKLOG)`, which asks Maude's socket manager to create a server socket at port `PORT`, with `BACKLOG` a positive integer specifying the maximum allowed number of queue requests. The message elicits either a message`createdSocket(O, socketManager, SERVER-SOCKET)`, or a message `socketError(O, socketManager, S)`.
5. `acceptClient(SERVER-SOCKET, O)`, which causes Maude to listen for incoming connections at `SERVER-SOCKET`. If a client connection is accepted, Maude responds back with the message `acceptedClient (O, SERVER-SOCKET, ADDRESS, SOCKET)`, where `ADDRESS` is the client's address and `SOCKET` is a newly created socket for communicating with the client. The message `socketError(O, socketManager, S)` is issued in case of failure.
6. `closeSocket(SOCKET, O)`, which causes Maude to close the socket and issue the message `closedSocket(O, SOCKET, S)`.

Rewriting a term with external objects in Maude is performed with the `erewrite` command (also abbreviated as `erew`). A more detailed discussion of sockets and support for external objects in Maude can be found in [11].

4.2 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* (a term of the sort `Loc`), which is defined as a combination of an address (such as a URI or an IP address) and a port number. 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 loc : String Nat -> Loc .
  op s : Loc Nat -> XSOid .
  op e : Loc Nat -> EOid .
```

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 *local configuration* at some node, or simply a *configuration*, 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. Local configurations are encapsulated by an operator `op [_] : Configuration -> LocalSystem` to support managing time and its effects.

4.3 Sockets and Messaging

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 using services from sites as server objects. In particular, when an expression object `O` makes a site call with actual parameters `C` to an external site identified by `s(loc(SR, PT), N)`, a site call message of the form `s(loc(SR, PT), N) <- sc(O, C, H)` is created within the configuration, where `H` is a temporary handle that uniquely identifies the call. This message then triggers a request for creating a client socket to the called site through the following equation:

```
eq s(loc(SR, PT), N) <- sc(O, C, H)
   = < p(O, H) : Proxy | param : C, response : "" >
     createClientTcpSocket(socketManager, p(O, H), SR, PT) .
```

In addition, the equation creates a temporary *proxy* object identified by `p(O, H)`, 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 socket to `loc(SR, PT)` is successfully created, the message `createdSocket(OP, socketManager, SC)`, targeted to the proxy `OP`, is introduced into the configuration. This causes the proxy to send the site call message to the external site object through the socket `SC`, as specified by the rewrite rule (the variable `AS` denotes the rest of the attributes in the object):

```
rl [SendExtCall] :
  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 that 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`.

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 reports the error by issuing the `socketError(OP, socketManager, S)` message. 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(SC, OP)` message.

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

When some string `S` is received through the socket, the message `received(OP, OD, S)` appears, and the proxy object stores `S` in its buffer and waits for further input.

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

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

```
rl [ProcessExtResponse] :
  closedSocket(p(O,H), SC, S)
  < p(O,H) : Proxy | response: S' , AS >
  => O <- 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. Once a client has been connected, the site receives the request (which contains the actual parameters for the site call), processes it, and, if appropriate, emits a response back to the client and closes the socket. Site objects employ a similar mechanism for message handling as the one described above.

It is worth noting that, just like any other distributed communication mechanism, messaging through sockets is inherently prone to various potential communication problems, including connection initiation errors, 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.

4.4 Timed Behavior

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*, where each node in the distributed configuration maintains its

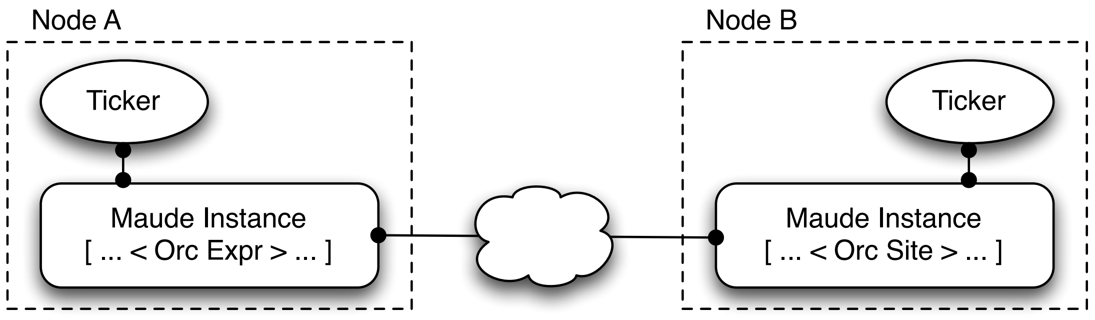


Figure 3: A diagram illustrating the general structure of a distributed Orc configuration. Dashed rectangles represent node boundaries, solid rounded rectangles represent local configurations, and darkened circles represent endpoints of TCP sockets.

local clock, which also emphasizes Orc's philosophy of having the communicating entities as loosely coupled as possible. Furthermore, for all the applications we have so far specified in Orc, distributed clock synchronization mechanisms, such as Lamport counters [20] or vector clocks [21, 22], are not required for programs correctness. This is primarily due to the fact that in most applications clocking information is used either locally (for example with the local *rtimer* site) or to time incoming responses. 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 from a *ticker* object external to Maude, but with access to the node's real-time system clock. Thus, for each node in the distributed configuration, the clock of the local Orc configuration is indirectly managed by a corresponding ticker object within the same node. It is important to note that, although we use sockets to implement it, the ticker object is local to the node and is thus guaranteed to provide fairly accurate clocking information. Figure 3 illustrates schematically the deployment architecture of a distributed Orc configuration with timing.

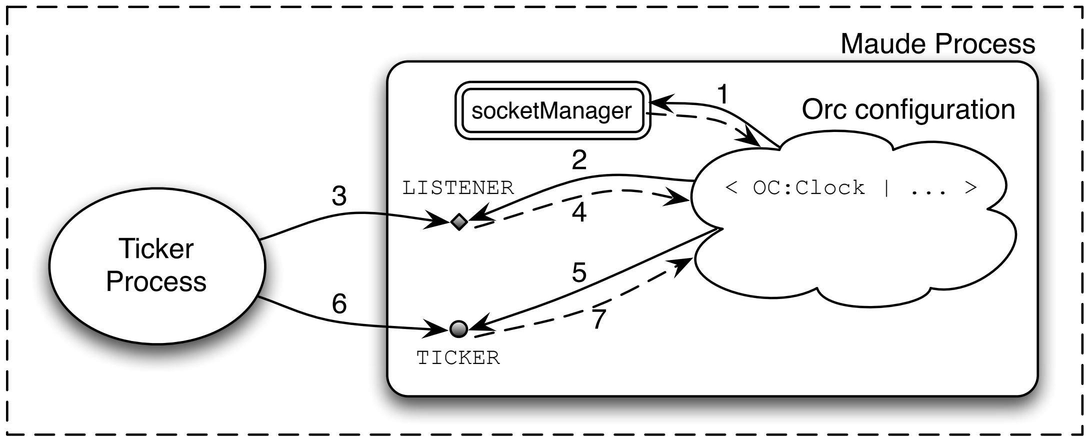
The diagram in Figure 4 outlines the steps involved in initializing a connection with the co-located ticker object and receiving the first clock tick.

Upon initialization, the clock object within an Orc configuration requests a server socket, by issuing the message `createServerTcpSocket(...)`, to be used for listening for a connection from the local ticker process, which is a Java process that is run in every node of a distributed configuration. The ticker process uses the built-in Java classes `Timer` and `Socket` to generate and send a tick message every t milliseconds to its corresponding Maude process, where t is a positive integer value. The clock object waits for a connection as soon as the server socket is created:

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

where `LISTENER` is the newly created clock server socket. Once the ticker object is connected, the message `acceptedClient(OC, LISTENER, IP, TICKER)` appears, with `IP` the originating address of the ticker object, and `TICKER` the newly created client socket for communicating with the ticker object. This causes the clock object to become ready for incoming clock ticks according to the following rule:

```
rl [InitClockSocket2] :
```



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)`

Figure 4: Establishing a connection with the ticker object and receiving clock ticks.

```
  acceptedClient(OC, LISTENER, IP, TICKER)
  < OC : Clock | AS >
   => < OC : Clock | 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 [17] for an explanation of the *delta* methodology).

```
crl [tick] :
  [received(OC, TICKER, S) < OC : Clock | clk: c(N) > CF]
   => [< OC : Clock | clk: c(N + 1) > receive(TICKER, OC) delta(CF)]  if ...
```

The variable `CF` denotes the rest of the local configuration. Recall that the operator `[_]` encapsulates the local configuration. The process of receiving and processing time ticks repeats as long as the ticker object is supplying ticks through the clock socket.

An important observation 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 our distributed implementation, 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 local configuration for the *Posting* expression object has the form:

```
[ <>
  < C : Clock | clk : c(0) >
  createServerTcpSocket(socketManager, C, 54200, 10)
  < e(loc("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 local 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(loc("10.0.0.4", 44800), 0) : Site |
      name : 'seller, state : (item(1910, 5, 500), item(1720, 7, 700)) , ... >
  createServerTcpSocket(socketManager, s(loc("10.0.0.4", 44800), 0), 44800, 10)]
```

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.

Each local configuration in `Dist-Auction` may run on a different node in a communication network. The diagram in Figure 5 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 orc-distributed.maude auction-manager.maude
```

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

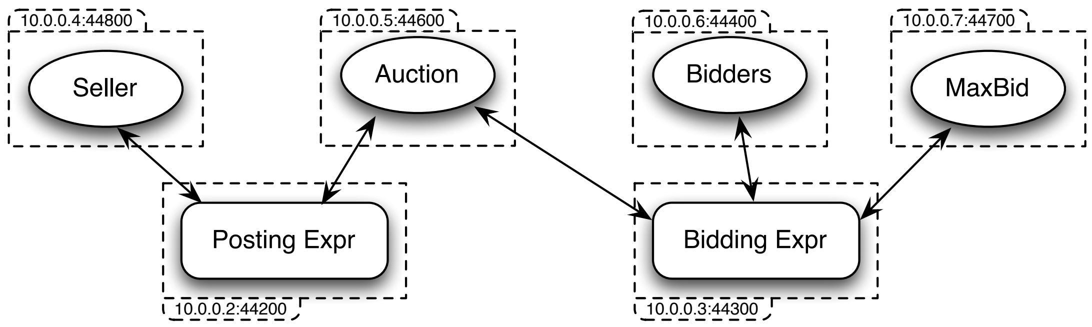


Figure 5: The deployment architecture of the `Dist-Auction` Orc program

```
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 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 auctioned 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 implementation described above is very useful in prototyping and deploying Orc programs as it allows observing actual possible behaviors in realistic environments. However, the implementation technique 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 use of TCP sockets and ticker objects, which are outside the scope of the Maude formal analysis tools. 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 this section, is to define rewriting logic specifications in Real-Time Maude for both Maude sockets and externally defined tickers, so that the distributed implementation can be turned, with minimal effort, into a formally analyzable specification.

5.1 Formal Specification of TCP Sockets

Maude's TCP sockets can be formally specified by defining abstractions of Maude instances, sockets, and their behaviors. We develop a rewriting specification $\mathscr{R}_{Socket}$ of sockets, which is based on previous work in [23, 24]. The specification abstracts Maude instances with objects of the class *Process*, and server and client sockets with objects of *ServerSocket* and *Socket* classes, respectively, which mediate communication between processes. A process object has the form $\langle PID : Process \mid sys : S \rangle$, with *PID* an object identifier and *S* an encapsulated local configuration. A client socket object of the form $\langle SID : Socket \mid endpoints : [PID_a, PID_b] \rangle$ abstracts a bidirectional client TCP socket set up between processes PID_a and PID_b, while a server socket object simply maintains the server's address and port: $\langle SID : ServerSocket \mid address : A, port : N \rangle$. Server socket objects are created using a *Manager object*, $\langle socketManager : Manager \mid counter : N \rangle$, abstracting Maude's socket manager and maintaining a counter for generating fresh socket object identifiers. Finally, messages in $\mathscr{R}_{Socket}$ have formats that are almost identical to those used by Maude sockets.

Different useful abstractions of socket behaviors can be defined. For the formal model of DIST-ORC, we choose an abstraction level that captures most interesting behaviors and yet can be efficiently analyzed. The abstraction essentially considers potential client socket creation errors and a somewhat limited form of communication delays and failures. This design choice abstracts away uninteresting behaviors, such as server socket creation problems, and approximates actual messaging problems, such as unavailable or unreachable servers, and unreliable networks. The main features of $\mathscr{R}_{Socket}$ are explained below.

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

```
rl [CreateServerTcpSocket] :
  < PID : Process | sys : [createServerTcpSocket(socketManager, O, PT) 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 : PT > .
```

The rule creates a server socket object, identified by `server(N)`, 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', SR, PT)`, 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', SR, PT) CONF' ] >
  < socketManager : Manager | counter : M >
  < server(N) : ServerSocket | address : SR, port : PT >
  => < PID : Process | sys : [acceptedClient(O, server(N), SR, socket(M)) CONF] >
     < PID' : Process | sys : [createdSocket(O', socketManager, socket(M)) CONF'] >
     < socketManager : Manager | counter : (M + 1) >
     < server(N) : ServerSocket | address : SR, port : PT >
     < socket(M) : Socket | endpoints : [PID : PID'] > .
```

In this rule, the server is in a state accepting incoming connections from clients, specified by matching a server at address and port `SR:PT` 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 the following rule, where the client process gets the `socketError(O', socketManager, "")` message from the socket manager, and no new socket object is created.

```
rl [CreateClientSocketFail] :
  < PID : Process | sys : [acceptClient(server(N), O) CONF] >
  < PID' : Process | sys : [createClientTcpSocket(socketManager, O', SR, PT) CONF' ] >
  < socketManager : Manager | counter : M >
  < server(N) : ServerSocket | address : SR, port : PT >
  => < PID : Process | sys : [acceptClient(server(N), O) CONF] >
     < PID' : Process | sys : [socketError(O', socketManager, "") CONF'] >
     < socketManager : Manager | counter : M >
     < server(N) : ServerSocket | address : SR, port : PT >  .
```

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.

```
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(O, SOCKET)` message, acknowledging the send action to the sender, and a `received(O', SOCKET, C, R)` message, signaling *delayed* delivery of the sent message to the receiver *in R time units*. The delay value for a transmitted message is non-deterministically extracted using a matching equation in the condition from a finite, non-empty set of delays `DS` maintained by a special object `DID`. A delay set for a given distributed Orc application can be specified as part of its initial state. 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, 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

Time and its effects on the distributed Orc configuration are formally specified using the standard and general technique of capturing logical time in real-time rewrite theories [25], and facilitated by Real-Time Maude [19]. Essentially, the time domain is represented by a sort *TimeInf* (time with infinity), and a *global tick* rewrite rule is used to synchronously advance time and propagate its effects across the *encapsulated global configuration*, a term of the sort *GlobalSystem*, of the form $\{Conf\}$, where *Conf* is the Orc configuration consisting of all process and socket objects. Furthermore, the tick rule, which plays the role of the ticker objects in the distributed implementation, is defined globally as follows (with `R'` a variable ranging over the positive rational numbers):

```
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 clock objects and updates time delays in all site calls and returns 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(< O : Clock | clk : c(R) > CF, R')
   = < O : 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. We assume a general *maximal* strategy that in each tick advances time by the *maximum time elapse*, which is defined by the function `mte` as the minimum delay across all site call messages and returns in the global Orc configuration. This 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 [8, 9], the tick rule is also made conditional on the fact that no other instantaneous transition is possible. This is precisely captured by the `eagerEnabled` predicate, which is true on configurations with enabled internal transitions or pending message exchanges. This imposes a precedence of rule application, where time ticks have the lowest priority among all transitions.

It is important to note 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 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 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 properties about distributed applications in Orc. To illustrate this formal verification capability, we use Real-Time Maude to formally analyze the distributed implementation `Dist-Auction` of the auction case study. In particular, we perform time-bounded linear temporal logic model checking with commands of the form (mc *term* |=t *formula* in time <= *timeLimit* .), and timed search using (find earliest *term* =>* *pattern* such that *condition* .), which finds a state reachable within the shortest possible time that matches the given pattern and satisfies the given condition. Verification is applied on a *closed* system specification that includes definitions of all required sites (servers) and expressions (clients) in the AUCTION application.

In 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 units, respectively. The function 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 the singleton set $\{0.1\}$, unless otherwise indicated. The atomic predicates used are:

1. *commError*, which is true in states with communication errors:

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

2. *sold*(id), which is true in states where the item *id* has been sold:

```
op sold : Nat -> Prop .
eq {< PID : Process |
        sys : [< O : XSite | name : 'auction,
                             state : won(winner(N, id, M), WN) OST > CF] > CF'}
    |= sold(id) = true .
```

where the term winner(N, id, M) matches a winning bid M on item id by the Nth bidder.

3. *hasBid*(id), which is true when the item *id* has been bid on:

```
op hasBid : Nat -> Prop .
eq {< PID : Process |
        sys : [< O : XSite | name : 'bidders,
                             state : bidders(b(N, [id, M] IBS) BS) OL > CF] > CF'}
    |= hasBid(id) = true .
```

where the term b(N, [id, M] IBS) matches a bid M on item id by the Nth bidder.

4. *conflict*(id), which is true when item *id* has two different winners:

```
op conflict : Nat -> Prop .
eq {< PID : Process |
        sys : [< O : XSite | name : 'auction ,
                             state : won(winner(N, id, M), winner(N', id, M'), WN)
                                     OST > CF] > CF'}
   |= conflict(id) = true .
```

A property that is typically required in an auction management system is that an item with at least one bid is eventually sold: $\Box \bigwedge_i (\mathit{hasbid}(id_i) \rightarrow \Diamond \mathit{sold}(id_i))$. This can be shown to be guaranteed by Dist-Auction in the absence of communication problems and excessively large delays. The property itself is specified in Real-Time Maude as the following formula commitAllNoErrors (with ~ denoting the LTL negation operator) :

```
op commit : Nat -> Formula .
eq commit(id) = hasBid(id) -> <> sold(id) .
op commitAllNoErrors : -> Formula .
eq commitAllNoErrors = ([] ~ commError) -> [] (commit(1910) /\ commit(1720)) .
```

The property is then verified with the time-bounded model checking command:

```
Maude> (mc initial(1/10) |=t commitAllNoErrors in time <= 15 .)
rewrites: 7052663 in 14413ms cpu (14420ms real) (489317 rewrites/second) ...
Result Bool : true
```

The property is satisfied with a communication delay of 0.1 time units. In fact, the property is satisfied when communication delays are bounded 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 but with `initial(1/4)`.

Another property an auction management system must guarantee is that every item sold has a unique winner: $\Box \bigwedge_i \neg \mathit{conflict}(id_i)$. This property can be shown satisfiable in `Dist-Auction` regardless of communication errors. The property is specified in Real-Time Maude as the formula `uniqueWinnerAll`:

```
op uniqueWinner : Nat -> Formula .
eq uniqueWinner(id) = ~ conflict(id) .
op uniqueWinnerAll : -> Formula .
eq uniqueWinnerAll = [] (uniqueWinner(1910) /\ uniqueWinner(1720)) .
```

The property is verified by the following command :

```
Maude> (mc initial(1/10) |=t uniqueWinnerAll in time <= 15 .)
rewrites: 8613539 in 19627ms cpu (19800ms real) (438843 rewrites/second) ...
Result Bool : true
```

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

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, and the anonymous reviewers and Peter Ölveczky for helpful comments on an earlier version of the paper. 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., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Journal of Software and Systems Modeling **6**(1) (March 2007) 83–110

[2] Hoare, T., Menzel, G., Misra, J.: A tree semantics of an orchestration language. Engineering Theories of Software Intensive Systems (2005) 331–350

[3] 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

[4] 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

[5] Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: A timed semantics of Orc. Theor. Comput. Sci. **402**(2-3) (2008) 234–248

[6] 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

[7] Cook, W.R., Misra, J.: Implementation outline of Orc. `http://orc.csres.utexas.edu` (2005)

[8] 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

[9] AlTurki, M., Meseguer, J.: Reduction semantics and formal analysis of Orc programs. Electron. Notes Theor. Comput. Sci. **200**(3) (2008) 25–41

[10] AlTurki, M., Meseguer, J.: Rewriting logic semantics of Orc. Technical Report UIUCDCS-R-2007-2918, University of Illinois at Urbana Champaign (November 2007)

[11] 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)

[12] 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)

[13] 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

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

[15] 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

[16] Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3) (2007) 213–237

[17] Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2) (2007) 161–196

[18] Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. Electron. Notes Theor. Comput. Sci. **176**(4) (2007) 5–27

[19] Ölveczky, P.C.: Real-Time Maude 2.3 manual (August 2007) `http://heim.ifi.uio.no/~peterol/RealTimeMaude/`.

[20] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565

[21] 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.).

[22] 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

[23] Durán, F., Riesco, A., Verdejo, A.: A distributed implementation of Mobile Maude. Electron. Notes Theor. Comput. Sci. **176**(4) (2007) 113–131

[24] 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

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