Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Executable rewriting logic semantics of Orc and formal analysis of Orc programs

Musab A. AlTurki[a,*,1], José Meseguer[b,2]

[a] King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[b] The University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

ARTICLE INFO

Article history:
Received 8 November 2013
Received in revised form 29 January 2015
Accepted 26 March 2015
Available online 3 April 2015

Keywords:
Rewriting logic
Orc
Executable semantics
Maude
Formal analysis
Service orchestration

ABSTRACT

The Orc calculus is a simple, yet powerful theory of concurrent computations with great versatility and practical applicability to a very wide range of applications, as it has been amply demonstrated by the Orc language, which extends the Orc calculus with powerful programming constructs that can be desugared into the underlying formal calculus. This means that for: (i) theoretical, (ii) program verification, and (iii) language implementation reasons, the *formal semantics* of Orc is of great importance. Furthermore, having a semantics of Orc that is *executable* is essential to provide: (i) a formally-defined *interpreter* against which language implementations can be validated, and (ii) a (semi-)automatic way of generating a wide range of *semantics-based* program verification tools, including model checkers and theorem provers.

This work proposes a *formal executable semantics* for Orc in rewriting logic, to support formal verification of Orc programs and to make possible semantics-based correct-by-construction Orc implementations. While being a very simple calculus, Orc has a quite *subtle* semantics, so that fully capturing all its semantic aspects is highly nontrivial. The two main sources of subtlety are: (i) its real-time semantics, and (ii) the priority of *internal* computations within an Orc expression over *external* computations that process responses from *external sites*. In this paper, we show a simple and elegant way of handling these two sources of subtlety in rewriting logic using an order-sorted type system supporting subtypes and subtype polymorphism, and "tick" rewrite rules for capturing time. Moreover, our rewriting semantics incorporates useful *semantic equivalences* between Orc programs as equations and equational attributes, making the semantics both more abstract and more efficient. The semantics of Orc is given in two different styles: (i) an *SOS style*, which is directly based on the original SOS of Orc, whose correctness follows immediately by construction, and (ii) a *reduction semantics*, which is much more efficiently executable and analyzable, as shown through several experiments, and whose correctness is proved using a strong bisimulation theorem. The paper also presents MOrc, a simulator and model checking tool based on the rewriting semantics of Orc and Real-Time Maude. MOrc facilitates formal verification of Orc programs, and allows for user-defined state predicates and LTL formulas, with no need for any prior knowledge of Maude or its rewriting logic foundations.

* Corresponding author.
E-mail addresses: musab@kfupm.edu.sa (M.A. AlTurki), meseguer@cs.illinois.edu (J. Meseguer).
[1] Supported by KFUPM Grant JF121005.
[2] Supported by AFOSR Contract FA8750-11-2-0084 and NSF Grants CCF 09-05584 and CNS 13-19109.

http://dx.doi.org/10.1016/j.jlamp.2015.03.003
2352-2208/© 2015 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

The Orc concurrency calculus [59,61,82] is very remarkable in that it combines great simplicity and mathematical elegance with great versatility and practical applicability to a very wide range of concurrent programming [61,42], web-based programming [61], business processes [22], and distributed cyber-physical system applications. Indeed, the great elegance and naturalness with which applications in all these areas can be programmed has been amply demonstrated by the Orc language [41,60], which extends the Orc calculus with powerful programming constructs. The Orc language's relationship to the Orc calculus can be viewed as the analogue of the relationship between a purely functional language and the lambda calculus: in both cases, all the language's constructs can be desugared into the underlying formal calculus, which is crucial for both ease of reasoning and ease of developing correct implementations.

All this means that for: (i) theoretical, (ii) program verification, and (iii) language implementation reasons, the *formal semantics* of Orc is of great importance. Furthermore, there is by now overwhelming evidence in various approaches to formal semantics that the by far most useful semantic definitions are *executable formal semantic definitions*, for two main reasons. First, many complex languages do not have a formal semantics at all, but have at best lengthy and ambiguous standards in natural language and various compilers, often exhibiting different behaviors. In such a case, a "paper semantics" is of very limited use, since the language's complexity makes it in practice virtually impossible to validate whether such a semantics (which cannot be compared to any other one by the language's lack of a formal semantics) really captures the intended, informal semantics described in its standard. By contrast, an executable formal semantics automatically provides a formally-defined *interpreter*, so that the semantic definition can be validated against the language's standard and against other mature language implementations to ensure that the informal semantics has been correctly formalized. Indeed, this is exactly the approach advocated in the rewriting logic semantics project [51,56,73,57], where large languages have been given an executable semantics in rewriting logic, including Java and the JVM [27,28], Scheme [49], Verilog [50], and, more recently, C [26]. The case of C is a good illustration of why an efficient executable semantics is essential, since the work in [26] is the first formal semantics ever given of the entire C language, a semantics which has been validated against the entire gnu C compiler torture suite and has uncovered bugs in several C tools.

A second reason why an executable formal semantics is enormously more useful than a "paper semantics" is that a wide range of program verification tools, including model checkers, [51,56,57], theorem provers [57,70] and static analysis tools [56,57] can be based *directly* on such an executable formal semantics in the sense of both being generated from the executable semantics and embodying such a semantics. This, in turn, has two main advantages: (i) using languages such as Maude [21] many of these program analysis tools can be generated automatically and for free from the semantic definition: for example, the JavaFAN model checkers for Java and the JVM are automatically generated that way and compare favorably with well-known Java model checkers [27,28], and (ii) there is no gap between the formal semantics and the program analysis tools, since the tools are based on the program semantics. This is by no means the case for most formal tools, which embody such a semantics only implicitly and sometimes erroneously. For example, the work in [26] has uncovered several semantic errors in well-known theorem provers for C, and the work in [50] uncovered similar problems in mature Verilog tools.

The goal of this work is to propose a *formal executable semantics* for Orc in rewriting logic [52], and to exploit such an executable semantics in the above-mentioned ways to support formal verification of Orc programs and to make possible semantics-based correct-by-construction Orc implementations. Compared with a language such as C, the task is in several ways much simpler, since Orc is a much simpler language than C, and has from the very beginning been designed as a formal calculus with an SOS semantics [61]. However, Orc, while being a very simple calculus, has a quite *subtle* semantics, so that fully capturing all its semantic aspects is highly nontrivial. In fact, it is not at all clear that this can be done in just any semantic framework. The sources of subtlety include the following:

1. **Real Time.** Orc is a real-time calculus, where the passage of time is essential and where sophisticated real-time concurrent applications can be developed. Any semantic framework not supporting such a real-time semantics will be useless.
2. **Internal vs. External Computation.** An Orc expression evaluates its constructs internally; but the evaluation may involve making calls to *external sites* and eventually receiving answers from such site calls, so that the internal evaluation can proceed. For example, an Orc expression may invoke both the CNN and BBC web sites with a given timeout, and then send zero, one, or two emails to a given user with the respective contents of the web sites which responded before the specified timeout. To avoid undesirable behaviors, internal computations should always be given priority over external ones. In [61], this is modeled by having *two transition relations* in Orc's SOS semantics, $\hookrightarrow_R$ and $\hookrightarrow_A$.

We show in Section 5 that these two sources of subtlety can be handled in a quite simple and elegant way by our rewriting logic semantics. Specifically, we show that using an order-sorted type structure [32], supporting subtypes and subtype polymorphism, completely solves subtlety (2), so that a single transition relation enforces the desired priority of internal over external computation. The solution of subtlety (1) is even simpler: the addition of a *single* additional "tick" rule, modeling time elapse, to the semantic rules describing Orc's instantaneous computations and the proper definition of time execution semantics and time delays are all that is needed to obtain a real-time semantics.

Another important feature of Orc's rewriting logic semantics is that it incorporates eleven useful *semantic equivalences* between Orc programs listed in Section 3.4; this makes the semantics both more abstract and more efficient. The point is that the rewriting semantics of Orc is a rewrite theory $(\Sigma_{Orc}, E_{Orc} \cup B_{Orc}, R_{Orc})$, where R_{Orc} are the transition rules, and $(\Sigma_{Orc}, E_{Orc} \cup B_{Orc})$ is an equational theory capturing the eleven semantic equivalences as five equational axioms ("structural equivalences") B_{Orc}, and six other equations E_{Orc}, which are confluent and terminating modulo B_{Orc}. Furthermore, all these semantic equations and rewrite rules satisfy all the required executability properties and are indeed *executable* in Maude [21] and its Real-Time Maude extension [64], giving rise to an Orc interpreter.

An executable formal semantics of Orc is useful both for formal analysis purposes, such as model checking Orc programs, and for deriving a correct-by-construction Orc implementation. However, for both these purposes executability of the rewrite theory expressing Orc's semantic definition is not enough: *efficient* executability is needed. That is, both a formal tool like an Orc model checker and a semantics-based Orc implementation, such as the one presented in [6], should have acceptable performance. Since our Orc rewriting logic semantics closely follows Orc's SOS semantics, the resulting rewrite theory, though executable and capable even of model checking nontrivial Orc programs, is however inefficient. The main reason is the chosen SOS format: since in small step SOS each inference rule may have one or more preconditions that are themselves transitions, the corresponding rewrite rule modeling such an inference rule must be *conditional*. Execution of conditional rewrite rules is intrinsically inefficient, both because of the recursive nature of the conditions, which may invoke other rules and conditions, and because of the non-deterministic nature of the transitions, which requires breadth first search to satisfy conditions.

Fortunately, as explained in [73], rewriting logic semantics can faithfully model many different definitional styles, so that a much more efficient semantic definition may be attained using a different style. This suggests using a *reduction semantics* style, where virtually all semantic rules will be unconditional. This is exactly the task undertaken in Section 6, where a reduction semantics for both Orc's synchronous SOS semantics and its real-time semantics by adding a single additional "tick" rule are developed. This of course raises the issue of the *equivalence* between the original small-step styled definition and new reduction semantics. This we prove in Section 6.5, in the form of a *strong bisimulation theorem* between the two semantic definitions. The reduction semantics is indeed much more efficient than the small step style one, as shown by the performance comparisons when executed in Maude, which are presented in Section 6.6.

As already mentioned, and as substantiated in Section 6.6, one key use of Orc's reduction style semantics is to obtain an *efficient* model checker for Orc. Since Real-Time Maude directly supports execution and model checking of invariants using the `trewrite` and `search` commands, and also LTL model checking, the Real-Time Maude implementation of the reduction style semantics already provides both an Orc simulator and an Orc model checker for free. However, an Orc user needs to have some familiarity with the underlying Real-Time Maude system to model check Orc programs. This is because, at the very least, the *state predicates* to be used in invariants and in LTL formulas need to be defined. To make the Orc model checker more easily usable, the MOrc tool described in Section 7 has been developed. MOrc is a web-based tool that allows the user to enter Orc programs to it, and also to define both state predicates and LTL formulas in an Orc-based manner, *with no need for any prior knowledge of the underlying Real-Time Maude* or its rewriting logic foundations. A user can then either execute an Orc program in MOrc or request from MOrc to model check any LTL formula or invariant. Internally, of course, what the MOrc tool does is to translate the Orc program and the given formula into their Real-Time Maude representation in the reduction style semantics, and then report back to the user the result of an execution or of a model checking command.

We finish the paper with a discussion of related work in Section 8, and with some concluding remarks in Section 9.

Compared with our previous work in [3,5], the work presented here provides new results and substantial improvements and extensions. Specifically, since the initial versions in [3,5], both the SOS-based and the reduction rewriting semantics of Orc have been thoroughly refined and extended to achieve a more complete, elegant, and efficiently executable specification. In particular, using order-sorted structures for Orc values and order-sorted declarations for Orc expressions and action labels, and using membership equations enables a simpler and more elegant specification of the synchronous semantics of Orc that can be executed and analyzed more efficiently than with just the many-sorted specifications used before. It also enables a concise representation of the new *otherwise* Orc combinator and its semantics. Furthermore, unlike previous versions, which were restricted to modeling discrete time domains, the semantics is now capable of handling dense time domains, using ideas from real-time rewrite theories [67] and Real-Time Maude [64], with implementations in both (Core) Maude and Real-Time Maude. Moreover, the rewriting semantics specification we present here incorporates as equational properties some fundamental algebraic laws about Orc expressions, making the semantics both more abstract and efficient. We also provide detailed proofs showing that both rewriting semantics specifications satisfy some desirable executability properties. As a side-effect to these improvements to the rewriting semantics specification, we present a set of SOS rules defining a self-contained transition relation specifying the synchronous semantics of Orc, in which no external rule application constraints are necessary, by exploiting the connection between order-sorted theories and context-free grammars. Finally, we describe a new web-based tool, MOrc, based on the reduction rewriting semantics specification of Orc, which enables expressive formal analysis of Orc programs, while requiring minimal knowledge of the internal Maude representation.

2. Preliminaries on rewriting logic and Maude

Rewriting logic [52] 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 [56,73,57,67] and references there). The unit of specification in rewriting logic is a *rewrite theory*, which gives a formal description of a concurrent system including its static state structure and its dynamic behavior. Assuming that t, u, v and w (and their decorated variants) are terms and s is a sort, a rewrite theory, in its most general form, is a tuple $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, consisting of: (i) a theory $(\Sigma, E \cup B)$ in membership equational logic (MEL) [53], where Σ is a MEL signature having a set of kinds, a family of sets of operators, and a kind-indexed family of disjoint sets of sorts, E is a set of Σ-sentences, which are universally quantified Horn clauses with atoms that are either equations $(t = t')$ or memberships $(t : s)$, and B is a set of equational axioms, such as commutativity, and/or associativity and/or identity axioms for some operators in Σ; (ii) a set R of universally quantified labeled *conditional rewrite rules* of the form:

$$(\forall X)\, r : t \to t' \textbf{ if } \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_l w_l \to w_l' \qquad (\star)$$

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

While the MEL theory $(\Sigma, E \cup B)$ specifies the user-defined syntax and equational axioms, which define the *system states* as elements of the initial algebra associated to $(\Sigma, E \cup B)$, a rule $r : t \to t'$ **if** C in R gives a general pattern for a possible *concurrent transition* in its state (modulo the restrictions imposed by ϕ), with the intuition that an instance $\theta(t)$ of t (with θ a substitution) may rewrite to $\theta(t')$ in the state of the system whenever the condition $\theta(C)$ is satisfied. Such rewrites are deduced according to the inference rules of rewriting logic, which are described in detail in [18]. Using these inference rules, a rewrite theory $\mathcal{R}$ proves a statement of the form $(\forall X)\, t \to t'$, meaning that, in $\mathcal{R}$, any instance of the state term t can *reach* the corresponding instance of the state term t' in a finite number of, possibly concurrent, steps. A detailed discussion of rewriting logic as a unified model of concurrency and its inference system can be found in [52] (see also the survey [54]). [18] gives a precise account of the most general form of rewrite theories and their models.

2.1. Real-time rewrite theories

A real-time rewrite theory [67] extends an ordinary rewrite theory with support for modeling real-time behaviors of systems. In particular, in a real-time rewrite theory $\mathcal{R}^\tau = (\Sigma^\tau, E^\tau \cup B^\tau, R^\tau, \phi)$: (i) the equational theory $(\Sigma^\tau, E^\tau \cup B^\tau)$ contains a sort for *Time* specifying the time domain, which can be either dense or discrete, and declares a system-wide operator that encapsulates the whole system being modeled into a special sort *GlobalSystem* for managing time elapse, and (ii) the set of rewrite rules R^τ is the disjoint union of two sets R_I and R_T, where R_I consists of *instantaneous* rewrite rules having the form $(\star)$ above and representing instantaneous transitions in the system, and R_T consists of *tick* rewrite rules modeling system transitions that take a non-zero amount of time to complete. A tick rewrite rule has the following form

$$r : \{t\} \xrightarrow{\tau} \{t'\} \textbf{ if } C$$

where τ is a term of sort *Time* representing the duration of time required to complete the transition specified by the rule. The global operator {_} encapsulates the whole system into the sort *GlobalSystem* to ensure the correct propagation of the effects of time elapse to every part of the system. A detailed discussion of real-time rewrite theories and their semantics, including a detailed explanation of how they can be reduced to ordinary rewrite theories by explicitly introducing a global clock as part of the global state, can be found in [67,64].

2.2. Maude and Real-Time Maude

Maude [21] is a high-performance implementation of rewriting logic and its underlying MEL sublogic. A basic unit of specification in Maude can be either a *functional module*, corresponding to a MEL theory $\mathcal{E} = (\Sigma, E \cup B)$, or a *system module*, defining a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$. A functional module may contain module inclusion assertions, sort and subsort declarations, operator symbols declarations (optionally with some equational attributes, including equational axioms B such as associativity, commutativity and/or identity), and conditional equations and membership axioms. *Admissible* functional modules, which are modules that satisfy some reasonable executability requirements, including ground confluence and termination (modulo the axioms B) and sort-decreasingness of the equations, can be executed in Maude by *equational simplification modulo axioms* using the equations E as simplification rules from left to right and Maude's matching algorithms modulo B to simplify a term to its canonical form with a least sort. Equational simplification modulo axioms of an admissible functional module yields an operational semantics, defined by the algebra of canonical forms $Can_{\Sigma/E,B}$, for its corresponding theory that coincides with its mathematical, initial algebra semantics, given by the initial algebra $T_{\Sigma/E \cup B}$ (see Sections 4.6–4.8 in [21] and cited references there). Simplification modulo axioms B can be performed by the `reduce` command in Maude.

An admissible system module, which may additionally contain possibly conditional rewrite rules, must satisfy the executability requirements for its equational part in addition to the ground coherence of the rules R with respect to the

$$E \in \mathit{ExpressionName} \qquad x \in \mathit{Variable} \qquad w \in \mathit{Value} \cup \{\textbf{stop}\}$$

$$\begin{array}{rcl}
\text{Orc program} & ::= & \vec{d}\ ;\ f \\
d \in \text{Declaration} & ::= & E(\vec{x}) \triangleq f \\
f, g \in \text{Expression} & ::= & \mathbf{0} \ \mid\ p(\vec{p}) \ \mid\ E(\vec{p}) \\
 & \mid & f \mid g \ \mid\ f > x > g \ \mid\ g < x < f \ \mid\ f\ ;\ g \\
p \in \text{Parameter} & ::= & x \ \mid\ w
\end{array}$$

Fig. 1. Syntax of Orc.

equations in E modulo B, and to admissibility conditions on the rules, which ensure that all variables in the rules can be instantiated by (incremental) matching. Such admissible modules can be executed in Maude by rewriting with rules (abiding by the restrictions imposed by ϕ) and oriented equations modulo the axioms B, which in this case corresponds exactly to the mathematical semantics of $\mathcal{R}$, which rewrites with R modulo the equational theory $E \cup B$ (see Section 6.3 in [21] and [18,80]). Rewriting of system modules can be performed in Maude by means of the `rewrite` command, which applies a rule-fair strategy to explore a possible behavior of the system, or the `search` command, which explores the entire reachable state space of the system, to find states instantiating a given pattern and satisfying a given semantic condition, following a breadth-first strategy. Furthermore, Maude provides a linear temporal logic (LTL) model checker for verifying safety and liveness properties.

While real-time rewrite theories with deterministic tick rules can be specified in Maude and analyzed using its standard analysis tools, a more expressive and flexible implementation and analysis of such theories, for both discrete and continuous time domains, is provided by Real-Time Maude (RTM) [66], which is an extension to Maude written using its reflective features. RTM modules provide the data types, operators, and execution strategies that enable the specification of timed modules with built-in or user-defined time domains. Time tick rewrite rules are in general non-deterministic, since the amount of time τ by which a system may advance its clock may be non-deterministic. Therefore, tick rules are in general not directly executable, and, for this reason, RTM provides a number of *time sampling strategies*, such as the general *maximal* sampling strategy (which advances time until the next instant when some instantaneous rewrite rule becomes enabled), which can be used to execute timed modules. Furthermore, RTM comes equipped with a range of formal analysis tools for timed modules, including timed rewriting (the command `trewrite`), timed and untimed search (`tsearch` and `utsearch`), and time-bounded and time-unbounded LTL model checking (the command `mc`). A complete description of RTM and its formal analysis features can be found in [66].

3. The Orc theory

Orc [59,61] is a timed theory for orchestration of services. It provides an expressive and elegant programming model for describing timed, concurrent computations. A *site* in Orc represents a service, which may range in complexity from a simple function to a complex web search engine, depending on the orchestration problem. A site may also represent the interaction with a human being, most commonly within the context of business work flows [78]. A site, when called, may produce, or *publish*, 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 the site call $\mathit{CNN}(d)$ might not respond because of a network failure or it may choose to remain silent because of an invalid input value d. Site calls are *strict*, i.e., they have a call-by-value semantics.

Being a timed theory, different site calls in Orc may occur at different times. A site call may be purposefully delayed using the *internal* site $\mathit{Rtimer}(t)$, 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, $\mathit{Rtimer}(t)$ 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 $\mathit{if}(b)$ site, which publishes a signal if b is true and remains silent otherwise, (2) $\mathit{let}(\vec{x})$, which publishes a tuple of the list of values in $\vec{x}$, or the value of $\vec{x}$ itself if $|\vec{x}| = 1$, and (3) *Clock*, which publishes the current time value.

3.1. Syntax of Orc

An Orc *expression* describes how site calls (and responses) are combined in order to perform a useful computation. Orc expressions are built up from site calls using four combinators, which were previously shown in [61,41] to be capable of expressing a wide variety of timed, distributed computations succinctly and elegantly. The abstract syntax of Orc is shown in Fig. 1. We assume a syntactic category *Value* that contains not only standard Orc values, such as numeric and boolean values and the *signal* value, but also site names as a distinguished sub-category *SiteName* of values that can be called (i.e., $\mathit{SiteName} \subset \mathit{Value}$). We also assume a special site response value **stop**, which may be used to indicate termination of a site call without necessarily publishing a standard Orc value.

An Orc *program* consists of an optional list of declarations, giving names to expressions, followed by an Orc expression to be executed. An expression can be either: (1) the silent expression ($\mathbf{0}$), which represents a site that never responds;

(2) a parameter or an expression call having an optional list of actual parameters as arguments; or (3) the composition of two expressions by one of the following four composition operators:

Symmetric parallel composition, $f \mid g$, which 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, $f > x > g$, which executes f, and for each value w published by f creates a fresh instance of g, with x bound to w, and runs that instance in parallel with the current evaluation of $f > x > g$. For example, if $Email(x)$ is a site that sends an e-mail message with contents x to a fixed address a, then the expression $CNN(d) > x > Email(x)$ may cause a news page to be sent to a. If $CNN(d)$ does not publish a value, $Email(x)$ is never invoked. Similarly, the expression $(CNN(d) \mid BBC(d)) > x > Email(x)$ 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 g has published its first value, which is then bound to x in f. For instance, the expression $Email(x) < x < (CNN(d) \mid BBC(d))$ sends at most one message, 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.

Otherwise composition, $f \,;\, g$, which attempts to execute f to completion. If f terminates without ever publishing a value, g is then executed. Otherwise, if f publishes a value during its execution, g is ignored. For example, suppose CNN publishes a **stop** value when called with invalid date values. Then, if d is a valid date value, the composition $CNN(d) \,;\, Email(\mathsf{err\_msg})$ never invokes $Email$ and may publish the news page from CNN. Otherwise, if d is invalid, an e-mail is sent and the value published by $Email$ is the value published by the composition.

A variable x occurs *bound* in an expression g when g is the right (resp. left) subexpression of a sequential composition $f > x > g$ (resp. an asymmetric parallel composition $g < x < f$). If a variable is not bound in either of the two above ways, it is said to be *free*. We use the syntactic sugar $f \gg g$ (resp. $g \ll f$) for sequential composition (resp. asymmetric parallel composition) when no value passing from f to g takes place, which corresponds to x *not* being a free variable in g. To minimize use of parentheses, we assume the following precedence order (from highest to lowest): $\gg$, $\mid$, $\ll$, $;$.

3.2. Small examples

We now list a few example Orc expressions, borrowed from [61]. Many more examples and larger programs can be found in [61,39,22,42,60]. The Orc expression below specifies a timeout t on the call to a site M:

$$let(x) < x < M() \mid Rtimer(t).$$

Upon executing the expression, both sites M and $Rtimer$ are called. If M publishes a value w before t time units, then w is the value published by the expression. But if M publishes w in exactly t time unites, then either w or $signal$ is published. Otherwise, $signal$ is published.

Another example is the standard programming idiom of the two-branch conditional **if** b **then** f **else** g, which can be written in Orc as the expression $if(b) \gg f \mid if(\neg b) \gg g$. Given the behavior of the internal site if, only one of the expressions f and g is executed, depending on the truth value of b.

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

$$Metronome(t) \triangleq let(signal) \mid Rtimer(t) \gg Metronome(t).$$

The expression named $Metronome$ can be used to repeatedly initiate an instance of a task every t time units. For example, the expression $Metronome(10) \gg UpdateLocation()$ calls on the task of updating the current location of a mobile user every ten time units.

3.3. Operational semantics of Orc

A structural operational semantics for the instantaneous (untimed) behaviors of Orc was originally given by Misra and Cook [61]. Fig. 2 lists an updated set of small-step SOS rules, based on the original SOS specification, that includes rules for the semantics of the *otherwise* combinator and **stop** site responses. The semantics uses two forms of internal expressions to represent intermediate transitional steps in the execution of an Orc expression, namely "$!v$", which publishes the value $v \in Value$, and "$?h$", with h a *handle* name, which is used to uniquely identify an unfinished site call.

The SOS semantics specifies the possible behaviors of an Orc expression as a labeled transition system with four label schemes corresponding to four types of actions an Orc expression may take: (1) publishing a value, $!v$, (2) calling a site, $M\langle \vec{v}, h\rangle$, with h a fresh handle name uniquely identifying this site call instance, (3) making an unobservable transition, τ, which may represent an expression call or a substitution event, and (4) consuming a site response, $h?w$, with h the handle for the corresponding site call and $w \in Value \cup \{\textbf{stop}\}$. In Fig. 2, n ranges over labels for non-publishing events, namely labels of types (2)–(4), while l ranges over all labels. In addition to the SOS rules in Fig. 2, the SOS semantics assumes some structural properties of Orc expressions that will be discussed in Section 3.4.

$$\frac{h \text{ fresh}}{M(\vec{v}) \xhookrightarrow{M\langle \vec{v},h\rangle} ?h}\ (\textsc{SiteCall}) \qquad \frac{f \xhookrightarrow{l} f'}{f \mid g \xhookrightarrow{l} f' \mid g}\ (\textsc{Sym})$$

$$?h \xhookrightarrow{h?v} !v\ (\textsc{SiteRetV}) \qquad \frac{f \xhookrightarrow{!v} f'}{f > x > g \xhookrightarrow{\tau} (f' > x > g) \mid [v/x]g}\ (\textsc{Seq1V})$$

$$?h \xhookrightarrow{h?\mathbf{stop}} \mathbf{0}\ (\textsc{SiteRetN})$$

$$!v \xhookrightarrow{!v} \mathbf{0}\ (\textsc{Publish}) \qquad \frac{f \xhookrightarrow{n} f'}{f > x > g \xhookrightarrow{n} f' > x > g}\ (\textsc{Seq1N})$$

$$\frac{E(\vec{x}) \triangleq f \in D}{E(\vec{p}) \xhookrightarrow{\tau} [\vec{p}/\vec{x}]f}\ (\textsc{Def}) \qquad \frac{f \xhookrightarrow{!v} f'}{g < x < f \xhookrightarrow{\tau} [v/x]g}\ (\textsc{Asym1V})$$

$$\frac{f \xhookrightarrow{!v} f'}{f\ ;\ g \xhookrightarrow{!v} f'}\ (\textsc{OtherV}) \qquad \frac{f \xhookrightarrow{n} f'}{g < x < f \xhookrightarrow{n} g < x < f'}\ (\textsc{Asym1N})$$

$$\frac{f \xhookrightarrow{n} f'}{f\ ;\ g \xhookrightarrow{n} f'\ ;\ g}\ (\textsc{OtherN}) \qquad \frac{g \xhookrightarrow{l} g'}{g < x < f \xhookrightarrow{l} g' < x < f}\ (\textsc{Asym2})$$

Fig. 2. Instantaneous, asynchronous structural operational semantics of Orc.

Two important refinements to the SOS specifications that are of central relevance to this work were proposed. First, as discussed by Misra and Cook in [61], the SOS semantics is highly non-deterministic, allowing *internal* transitions within an Orc expression (value publishing, site calls, and τ transitions) and the *external* interaction with sites in the environment (through site return events) to be interleaved in any order. This high degree of non-determinism may be undesirable. For example, in the expression $let(x) < x < Rtimer(1) \gg N() \mid M()$, which is supposed to give M priority over N, the call to M may actually be delayed in this semantics, thus defeating the purpose of prioritizing it over the call to N. In order to rule out such undesirable behaviors, a *synchronous semantics* was proposed in [61] by placing further constraints on the application of SOS semantic rules. The synchronous semantics was arrived at by distinguishing between *internal* and *external* events, and splitting the SOS transition relation $\hookrightarrow$ into two sub-relations $\hookrightarrow_R$, and $\hookrightarrow_A$, and characterizing set-theoretically, the complementary subsets of expressions (quiescent vs. non-quiescent) to which they are respectively applied. In previous work [3], we have presented two different approaches, namely, strategy expressions and equational conditions, in which this splitting into $\hookrightarrow_R$ and $\hookrightarrow_A$ can be faithfully captured in a rewriting logic semantics of Orc by enforcing an execution strategy that gives transitions corresponding to internal actions precedence over the external site return action. In Section 4.2, we describe a third, typed approach, based on sorts and subsorts, that is both more elegant and, in practice, more efficiently executable than the two previous approaches just mentioned.

A second refinement of the Orc SOS, by Wehrman et al. [82], endowed the original SOS specification with timing semantics in a way similar to timed process algebras [12]. This was achieved mainly by refining the SOS transition relation into a relation on time-shifted Orc expressions and timed labels of the form (l,t), where t is the amount of time taken by a transition. In this extended relation, a transition step of the form $f \xhookrightarrow{(l,t)} f'$ states that f may take an action l to evolve to f' in time t, and, if $t \neq 0$, no other transition could have taken place during the t time period. To properly reflect the effects of time elapse, parts of the expression f may also have to be time-shifted by t. However, for simplicity of presentation, the semantics described in [82], abstracted away the non-publishing events as unobservable transitions and considered only the asynchronous semantics of Orc. Sections 5 and 6 present a rewriting logic approach to capturing timed behaviors of Orc expressions, which also takes into account the *synchronous* semantics of Orc as described above.

3.4. Some algebraic properties

Orc was shown to possess several desirable structural properties, either using bisimulations based on the original and timed SOS semantics [40,81], or, alternatively, using graph isomorphisms in a tree-based denotational semantics [37]. We focus our attention here on the subset of these algebraic properties shown in Fig. 3. Our choice of this subset is motivated by the fact that Eqs. (6)–(11) are confluent and terminating modulo the axioms (1)–(5), so that equality under (1)–(11) becomes *decidable* by rewriting. Furthermore, since a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has both rules R and equations E, so that states are equivalence classes modulo E, we can obtain a more abstract and more efficient rewriting logic semantics of Orc by adding Eqs. (1)–(11) to the set E of equations in the rewrite theory $\mathcal{R}$ axiomatizing Orc.

Associativity, commutativity and identity axioms for symmetric parallel composition were proved in [40,81,37]. Associativity and right identity axioms of the otherwise combinator can also be proved by strong bisimulation (see [7]), and its left identity is assumed as a structural equivalence rule that is required to achieve its intended semantics. Proofs of the identities (6) and (9)–(11) are trivial, since both sides of these identities have no behavioral transitions, and are, thus, strongly bisimilar. The remaining two laws, namely (7) and (8), are also easy to show, and their proofs are given in [7].

Other algebraic properties of Orc expressions, which were shown in [40,81,37], are not suitable for algebraic simplification purposes because, when viewed as equations, they either fail to satisfy executability requirements, such as confluence and/or coherence with the Orc semantic rules, or they do not necessarily compute simpler normal forms. In particular, al-

$$(f \mid g) \mid h = f \mid (g \mid h) \quad (1)$$

$$f \mid g = g \mid f \quad (2)$$

$$f \mid \mathbf{0} = f \quad (3)$$

$$(f\ ;\ g)\ ;\ h = f\ ;\ (g\ ;\ h) \quad (4)$$

$$f\ ;\ \mathbf{0} = \mathbf{0}\ ;\ f = f \quad (5)$$

$$\mathbf{0} > x > f = \mathbf{0} \quad (6)$$

$$f < x < \mathbf{0} = [\mathbf{stop}/x]f \quad (7)$$

$$!v\ ;\ f = \ !v \quad (8)$$

$$M(\vec{p}) = \mathbf{0} \text{ if } \mathbf{stop} \in \vec{p} \quad (9)$$

$$w(\vec{p}) = \mathbf{0} \text{ if } w \notin \mathit{SiteName} \quad (10)$$

$$!\,\mathbf{stop} = \mathbf{0} \quad (11)$$

Fig. 3. Some algebraic properties of Orc expressions.

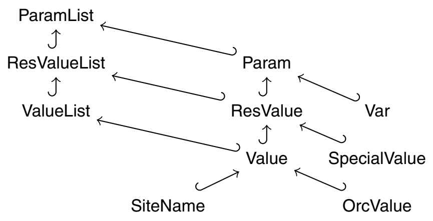


Fig. 4. Parameter subsort structure.

gebraic laws shown using weak bisimulations that ignore τ transitions, such as the law $f > x > let(x) = f$ [81], may break coherence of the semantic rules when used as equational properties, since they may cause an Orc expression to *miss* some behavioral transitions. Other identities may result in equations that are not confluent, such as the restricted left associativity law of sequential composition [40,81], where $FV(h)$ computes the set of free variables in h:

$$f > x > (g > y > h) = (f > x > g) > y > h \text{ if } x \notin FV(h)$$

(consider for example the term $f_1 > x > (f_2 > y > (f_3 > z > f_4))$, with $x \notin FV(f_3) \cup FV(f_4)$ and $y \notin FV(f_4)$). Finally, some identities, when used as oriented equations, may compute normal forms that are not necessarily structurally simpler than the original expressions, such as, for example, the law of distributivity of parallel composition over sequential composition [40,81]: $(f \mid g) > x > h = f > x > g \mid g > x > h$. For execution purposes, such equations add extraneous “simplification” steps that may adversely affect execution performance without actually arriving at simpler normal forms.

4. The Orc semantic infrastructure

The different styles of the rewriting logic semantics of Orc share a common infrastructure, which can be specified as a MEL sub-theory $\mathcal{R}_\Omega = (\Sigma_\Omega, E_\Omega \cup B_\Omega) \subset \mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}_\Omega \subset \mathcal{R}^{red}_{Orc}$ describing the structures for the semantic entities and the common behaviors that are needed for a complete specification of Orc's semantics. Below, we describe the most important components of the equational theory $\mathcal{R}_\Omega$, on which all later developments are based.

4.1. Parameters and substitution

We assume a sort Var for Orc variables. To account for substitution of variables with other parameters, we use the CINNI calculus of explicit substitution [74]. This is consistent with our choice of a first-order representation of Orc in rewriting logic and does not impair readability, since the CINNI notation is just a slight refinement of the usual textbook notation for higher-order syntax with explicit names. A more detailed discussion of the Orc instance of CINNI can be found in [7].

In addition, we assume a sort Param for Orc parameters, which, according to Orc's syntax in Fig. 1, are either variables of sort Var or site response values (including the special value **stop**) of the sort ResValue. Furthermore, response values other than **stop** are identified as either standard data types, such as integers and booleans, of sort OrcValue, or as site names of sort SiteName, which are values representing sites that can also be called. This classification of parameters is crucial to the semantics and is neatly captured by the subsorted structure illustrated in Fig. 4, in which a separate sort SpecialValue is used to represent **stop**, and three list super-sorts are declared.

4.2. Orc expressions

The set of Orc expressions that can be constructed from the syntax of Fig. 1, in addition to the internal publishing and handle expressions of the forms $!w$ and $?h$, is represented by a sort Expr, which is subsorted into the (singleton) zero expression subsort ZExpr, containing only $\mathbf{0}$ (which is declared as $\mathbf{0} : \to$ ZExpr), and the subsort of non-zero expressions, NZExpr. This distinction between $\mathbf{0}$ and other expression will simplify the specification and will help achieve a more efficiently executable semantics.

$$\begin{tikzcd}
 & & \mathsf{Expr} & \\
 & \mathsf{NZExpr} \arrow[ur, hook] & & \mathsf{ZExpr} \arrow[ul, hook'] \\
\mathsf{AExpr} \arrow[ur, hook] & & \mathsf{IExpr} \arrow[ul, hook'] &
\end{tikzcd}$$

Fig. 5. The subsort structure of Orc expressions.

In the synchronous semantics of Orc, the contrast between internal actions (publishing of values, site calls, and τ transitions) and the external action of a site return induces a corresponding distinction between expressions that can make an internal transition and others that cannot. To capture the synchronous semantics, we make this distinction explicit in the type structure by introducing the notions of *active* (AExpr) and *inactive* (IExpr) Orc expressions. Intuitively, an expression is active if it contains as a sub-expression a value publishing, a site call, or an expression call sub-expression that is *enabled*, and is inactive otherwise. This notion is made more precise in the following definition.

Definition 1 *(Active and inactive Orc expressions).* The set of *active* expressions $\mathcal{F}_a$ is the smallest set of non-zero expressions generated by the following rules:

1. $M(\vec{v})$, $E(\vec{p})$, and $!v$ are in $\mathcal{F}_a$.
2. If $f \in \mathcal{F}_a$, then $f > x > g \in \mathcal{F}_a$ and $f\ \mathbf{;}\ g \in \mathcal{F}_a$.
3. If $f \in \mathcal{F}_a$ or $g \in \mathcal{F}_a$, then $f \mid g \in \mathcal{F}_a$ and $g < x < f \in \mathcal{F}_a$.

A non-zero expression f is called *active* if $f \in \mathcal{F}_a$; otherwise, f is *inactive*.

Note that this notion of active expressions corresponds exactly to that of non-quiescent expressions in [61] (see Section 5.4 there). This notion can be elegantly captured in the type structure of the rewriting semantics by further subsorting NZExpr into two subsorts: AExpr, for active expressions, and IExpr for inactive expressions. The subsorting structure of Orc expressions is shown in Fig. 5. Since any non-zero Orc expression must either be active or inactive (and cannot be both), the subsorts partition the sort NZExpr. This is achieved by a combination of subsort-overloaded function symbol declarations for Orc's syntax, along with appropriate equational axioms and frozenness information, and a few simple membership axioms based on Definition 1, as we explain below.

Basic Orc expressions. An expression call $E(\vec{p})$, which is always active, has a corresponding declaration of the form $\_(\_) : \mathsf{ExprName} \times \mathsf{ParamList} \to \mathsf{AExpr}$, whereas a parameter call expression $p(\vec{p})$, which has the general declaration $\_(\_) : \mathsf{Param} \times \mathsf{ParamList} \to \mathsf{Expr}$, is active if and only if p is a site name $M \in \mathsf{SiteName}$ and $\vec{p}$ is a list of values $\vec{v} \in \mathsf{ValueList}$, and hence the subsort-overloaded declaration $\_(\_) : \mathsf{SiteName} \times \mathsf{ValueList} \to \mathsf{AExpr}$. For Inactive calls, which are calls that fail to satisfy the condition above (and are *not* semantically equivalent to $\mathbf{0}$), a third declaration $\_(\_) : \mathsf{Var} \times \mathsf{ValueList} \to \mathsf{IExpr}$ and two membership predicates

$$M(\vec{p}) : \mathsf{IExpr} \ \text{ if } \ \vec{p} \notin \mathsf{ValueList} \ \wedge \ \mathbf{stop} \notin \vec{p}$$

$$x(\vec{p}) : \mathsf{IExpr} \ \text{ if } \ \vec{p} \notin \mathsf{ValueList} \ \wedge \ \mathbf{stop} \notin \vec{p}$$

capture precisely when a parameter call is inactive. Given the parameter subsort structure in Fig. 4, this declaration and the two membership predicates define inactive parameter calls as those in which either: (1) the called parameter is a site name and the list of arguments contains at least one variable and no **stop** values, or (2) the called parameter is a variable and the argument list may contain variables or non-**stop** values. Note that, by identities (9) and (10) in the structural equivalence properties of Fig. 3, the other cases, in which the called parameter is a value or the argument list contains a **stop** value, are all semantically equivalent to $\mathbf{0}$, and are, therefore, of the sort ZExpr.

The other basic expressions, comprising handle expressions $?h$ and publishing expressions $!p$ are similarly specified. In particular, handle expressions $?h$ are always inactive and are simply specified by the declaration $?\_ : \mathsf{Handle} \to \mathsf{IExpr}$. Publishing expressions $!p$, which are active when $p \in \mathsf{Value}$ and inactive when $p \in \mathsf{Var}$, are specified by the following subsort-overloaded family of declarations:

$$!\_ : \mathsf{Param} \to \mathsf{Expr} \qquad !\_ : \mathsf{Value} \to \mathsf{AExpr} \qquad !\_ : \mathsf{Var} \to \mathsf{IExpr}$$

Note that the third case, when p is **stop**, is equivalent to $\mathbf{0}$, according to identity (11) in Fig. 3, and, therefore, has sort ZExpr.

Composed Orc expressions. To complete the specification of active and inactive expressions, function symbol declarations for the four Orc combinators are also subsort-overloaded according to Definition 1. Specifically, the symmetric parallel composition combinator has the following subsort-overloaded family of declarations (where the equational axioms of associativity, commutativity, and identity are specified by the equational attributes assoc, comm, id below, and the ditto keyword specifies the same equational attributes for other subsort-overloaded typings of the same operator):

$$\begin{array}{lll}
\_\,|\,\_ : & \mathsf{Expr} \times \mathsf{Expr} \to \mathsf{Expr} & [\mathsf{assoc\ comm\ id}: \mathbf{0}] \\
\_\,|\,\_ : & \mathsf{AExpr} \times \mathsf{Expr} \to \mathsf{AExpr} & [\mathsf{ditto}] \\
\_\,|\,\_ : & \mathsf{IExpr} \times \mathsf{IExpr} \to \mathsf{IExpr} & [\mathsf{ditto}]
\end{array}$$

which precisely state that a symmetric parallel composition is active if at least one of its subexpressions is active, and is inactive otherwise. Similarly, the following declarations specify the sequential composition operator:

$$\begin{array}{lll}
\_ > \_ > \_ : & \mathsf{Expr} \times \mathsf{Var} \times \mathsf{Expr} \to \mathsf{Expr} & [\mathsf{frozen}(3)] \\
\_ > \_ > \_ : & \mathsf{AExpr} \times \mathsf{Var} \times \mathsf{Expr} \to \mathsf{AExpr} & [\mathsf{ditto}] \\
\_ > \_ > \_ : & \mathsf{IExpr} \times \mathsf{Var} \times \mathsf{Expr} \to \mathsf{IExpr} & [\mathsf{ditto}]
\end{array}$$

Since the right subexpression of a sequential composition has no behavioral transitions, the sequential combinator symbol is declared frozen on its third argument (using the frozen attribute); i.e., we define $\phi(\_ > \_ > \_) = \{3\}$, so that no rewriting is allowed on the third argument. The declarations state that a sequential composition is active (resp. inactive) if and only if its left subexpression is active (resp. inactive). The operator declarations for the asymmetric parallel combinator $\_ < \_ < \_$ are similar to those of symmetric composition, while the declarations for the otherwise combinator $\_\,\mathbf{;}\,\ \_$ are similar to those of sequential composition, except that the symbol is declared frozen on its second argument, and with identity $\mathbf{0}$.

Algebraic properties. To fully account for the algebraic properties of Orc expressions, the semantic infrastructure includes equations that correspond to the algebraic identities (6)–(11) in Fig. 3. As mentioned above, identities (1)–(5) are specified as equational axioms of the respective Orc combinators, declared by the assoc, comm and id attributes in their operators' declarations.

4.3. Orc configurations

A state in the execution of an Orc program is defined by an Orc *configuration*, which (as in [62]) is a pair $\langle f, r\rangle$, where f is the Orc expression to be executed and r is a record structure consisting of five semantic fields: (1) a label field $lbl : l$, (2) an environment for expression names $env : \sigma$, (3) a pool of pending messages $msg : \rho$, (4) a set of currently used handle names $hdl : \eta$, and (5) a clock $clk : t$. A more detailed description of these fields follows.

Clock. Time is abstracted by the sort Time, which is specified as a totally ordered set with a least element zero. A supersort TimeInf of Time also includes ∞ as a top element, which is useful for specifying the proper timed semantics of Orc. An instantiation of the sort TimeInf can, therefore, be either discrete or dense. In our specifications we assume a dense time domain implemented by the non-negative rationals and maintained by the clock field in a configuration.

Environment. An *environment*, which is maintained to resolve references to expression names, is a term of sort Env, which is defined as a set of declarations (terms of sort Decl, with Decl < Env), formed with an associative and commutative set union operator $\_,\_$ with the empty set as its identity element. Initially, an environment is created out of the declaration list $\vec{d}$ of an Orc program $\vec{d}\,;\ f$ so that the following conditions hold: (1) a later declaration in the list $\vec{d}$ hides all previous declarations with the same expression name; and (2) all declarations in the resulting environment are visible to each other. This implies that an expression name has a unique defining declaration in an environment, and that (mutual) recursion is directly available.

Handles. A *handle* is a name of sort Handle that uniquely identifies a *pending* site call, which is a call awaiting a response from the environment. Since, by the SiteCall SOS rule of Fig. 2, fresh handle names need to be generated, a configuration maintains in its handles field a set η of currently used handles against which new names may be created. Sets of handles, of sort HandleSet, are constructed by an associative, commutative comma-denoted union operator, with the empty set of handles as its identity element.

Messages. We maintain a message pool (MsgPool) as a multiset of messages, constructed by the empty juxtaposition operator with the empty set as the identity element. A *message*, which is a term of the sort Msg, is either a site call message of the form $[M, \vec{v}, h]$, representing the parameters of the call along with a handle name identifying the site call, or a *simulated* site response of the form $[w, h]$, with w the site response corresponding to the call identified by h, that is waiting in the message pool to be consumed by the Orc expression. Although the environment in the SOS semantics of Orc is treated as a "black box" with unpredictable responses from remote sites, simulation of environment responses is needed to arrive at an executable specification. Simulation of responses is achieved by automatically converting a site call message $[M, \vec{v}, h]$ into a *potential* response message $[app(M, \vec{v}, \gamma), h]$, where *app* is a partial function that can simulate a response based on the intended semantics of M. It may also associate a delay γ to site responses. Such a potential response message cannot be processed until the delay is zero, at which point the message is replaced by a proper response message $[w, h]$, with w the site response evaluated by the function *app*. For instance, a site call message $[CNN, 1, h]$ is immediately converted into the potential response message $[app(CNN, 1, 5), h]$, with 5 a message delay. Then, after five time units have passed, the message becomes $[app(CNN, 1, 0), h]$, which, according to the semantics of site calls to *CNN* specified by the function *app*, is equivalent to the response message $[signal, h]$. The partial function *app* provides a flexible and modular mechanism for specifying delays and site call semantics [7].

Labels. The label field keeps track of the last event generated as a result of a configuration evolving into another, which is needed in the SOS semantics for inferring one-step transitions. To represent the four labels in the SOS rules in Fig. 2, we define a sort Label, and declare four operators of this sort:

$$\_\langle\_,\_\rangle : \mathsf{SiteName} \times \mathsf{ValueList} \times \mathsf{Handle} \to \mathsf{Label} \qquad \tau : \to \mathsf{Label}$$
$$\_?\_ : \mathsf{Handle} \times \mathsf{Value} \to \mathsf{Label} \qquad !\_ : \mathsf{Value} \to \mathsf{Label}.$$

We also use a special constant $\epsilon :\to \mathsf{Label}$ to represent absence of a label.

Therefore, the general form of an Orc configuration is: $\langle f,\ lbl : l \mid env : \sigma \mid msg : \rho \mid hdl : \eta \mid clk : t\rangle$. Given an Orc program $\vec{d}\ ;\ f$, its *initial* configuration, which can be constructed by an operator $[\_] : \mathsf{Program} \to \mathsf{Config}$, is of the form:

$$\langle f,\ lbl : \epsilon \mid env : \mathsf{init}(\vec{d}) \mid msg : \emptyset \mid hdl : \emptyset \mid clk : 0\rangle$$

where init is a function that initializes an environment structure from a list of declarations $\vec{d}$ according to the description given above.

Additionally, as part of the semantic infrastructure, we define two notions about Orc configurations (borrowed from Real-Time Maude [64]) that will be useful for defining the timed behaviors of Orc for both the SOS-based and the reduction rewriting semantics. The first is the notion of *eager* configurations, which are configurations that can make an instantaneous (internal or external) transition, i.e., configurations of the form $\langle f, r\rangle$ where either f is active or r has a pending site response that can be consumed. This notion is made more precise in the following definition, where $\hat{f}$ ranges over active Orc expressions and $\bar{f}$ over inactive expressions.

Definition 2 *(Eager Orc configuration).* An Orc configuration $\mathcal{C}$ is *eager* if $\mathcal{C}$ is of one of the following forms: (i) $\langle \hat{f}, r\rangle$; or (ii) $\langle \bar{f}, msg : \rho[w, h] \mid r\rangle$ with h a handle in $\bar{f}$.

This notion is easily captured by a (partial) predicate $\mathsf{eager} : \mathsf{Config} \to [\mathsf{Bool}]$ that evaluates to true if and only if the given configuration is eager using two equations corresponding to cases (i) and (ii) in Definition 2 above. The second notion is that of the *maximal time elapse* (or *mte*) of an Orc configuration, which specifies the maximum time shift until the next point in time when an instantaneous event (corresponding to the evaluation of an Orc expression as opposed to just advancing time on the configuration) may be enabled.

Definition 3 *(mte of an Orc configuration).* The maximum time elapse (mte) of an Orc configuration $\langle f,\ msg : \rho \mid r\rangle$ is the minimum time delay across all messages in ρ if ρ is non-empty, and is ∞ otherwise.

The time shift needed to advance the clock of an Orc configuration to the next point in time when an instantaneous action becomes enabled is determined by a function $\mathsf{mte} : \mathsf{Config} \to \mathsf{TimeInf}$, which is defined equationally according to the definition above.

5. The SOS-based rewriting semantics $\mathcal{R}^{sos}_{Orc}$

We now present an executable rewriting logic semantics of Orc that is based directly on the SOS semantics of Orc of Section 3.3. This semantics is obtained by mapping the SOS rules in Fig. 2 into a corresponding rewrite theory $\mathcal{R}^{sos}_{Orc} = (\Sigma^s, E^s \cup B^s, R^s, \phi^s)$ according to Meseguer and Braga's semantics-preserving transformation from Modular SOS [62] to rewriting logic. An initial version of the SOS-based rewriting semantics of Orc appeared in [3], where we described two different ways of capturing Orc's synchronous semantics: (1) strategy expressions, and (2) additional equational conditions. The semantics in [3] also captured timed behaviors in Orc, although timing, as specified there, was limited to discrete time domains, such as the natural numbers.

Since the initial version in [3], the rewriting semantics of Orc given by $\mathcal{R}^{sos}_{Orc}$ has been thoroughly refined and extended to achieve a more complete, elegant, and efficiently executable specification. First, using order-sorted structures for Orc values, a concise representation of the new *otherwise combinator* and its semantics has been achieved. Moreover, order-sorted declarations for Orc expressions and action labels, and membership equations enable a simpler and more elegant specification of instantaneous actions that can be executed and analyzed more efficiently than with just the many-sorted specifications used before. Furthermore, the semantics is now capable of handling dense time domains, using ideas from real-time rewrite theories [67] and Real-Time Maude [64], with implementations in both (Core) Maude and Real-Time Maude.

As discussed in Section 4, the rewrite theory $\mathcal{R}^{sos}_{Orc}$ extends the semantic infrastructure equational theory R_Ω, which captures the algebraic properties of Orc listed in Fig. 3 as equational axioms in B^s and equations in E^s. Below, we describe how $\mathcal{R}^{sos}_{Orc}$ captures the timed, synchronous semantics of Orc expressions, and discuss some of its important properties.

5.1. Instantaneous rewriting semantics rules

Since, by rewriting logic's **transitivity** inference rule, a rewrite computation $t \to t'$ may involve a sequence of one-step rewrites $t \to t_1 \to t_2 \to \cdots \to t_n \to t'$, we need to restrict rewrites of Orc configurations to be exactly one-step rewrites, corresponding to the single-step SOS behavior, as explained in [55]. For this purpose, we employ the SOS one-step modifier technique of [62,73], in which two operators are declared: (1) a (frozen) prefix dot operator $\cdot\_ : \mathsf{Config} \to \mathsf{Config}$, and (2) a non-frozen operator $\mathsf{smallstep} : \mathsf{Config} \to \mathsf{Config}$. By defining the rewrite rules that correspond to the SOS rules in Fig. 2 in the following format

$$\textsc{SiteCall}: \cdot\langle M(\vec{v}),\ lbl:l \mid msg:\rho \mid hdl:\eta \mid r\rangle$$
$$\rightarrow \langle ?h,\ lbl: M\langle \vec{v}, h\rangle \mid msg:\rho[M,\vec{v},h] \mid hdl:\eta,h \mid r\rangle \text{ if } h := \mathsf{fresh}(\eta)$$
$$\textsc{SiteRetV}: \cdot\langle ?h,\ lbl:l \mid msg:\rho[v,h] \mid hdl:\eta,h \mid r\rangle$$
$$\rightarrow \langle !v,\ lbl: h?v \mid msg:\rho \mid hdl:\eta \mid r\rangle$$
$$\textsc{SiteRetN}: \cdot\langle ?h,\ lbl:l \mid msg:\rho[\mathbf{stop},h] \mid hdl:\eta,h \mid r\rangle$$
$$\rightarrow \langle \mathbf{0},\ lbl: h?\mathbf{stop} \mid msg:\rho \mid hdl:\eta \mid r\rangle$$
$$\textsc{Publish}: \cdot\langle !v,\ lbl:l \mid r\rangle \rightarrow \langle \mathbf{0},\ lbl: !v \mid r\rangle$$
$$\textsc{Def}: \cdot\langle E(\vec{p}),\ lbl:l \mid env:\sigma, E(\vec{x}) \triangleq f \mid r\rangle \rightarrow \langle [\vec{p}/\vec{x}]f,\ lbl:\tau \mid env:\sigma, E(\vec{x}) \triangleq f \mid r\rangle$$

Fig. 6. Rewrite rules in $\mathcal{R}^{sos}_{Orc}$ for basic expressions.

$$\cdot\langle f, r\rangle \rightarrow \langle f', r'\rangle \text{ if } \bigwedge_{i=1}^{n} \cdot\langle f_i, r_i\rangle \rightarrow \langle f_i', r_i'\rangle \wedge C$$

with C an equational condition, we effectively restrict rewriting to single steps using the equation:

$$\mathsf{smallstep}(\langle f,\ lbl:l \mid r\rangle) = \mathsf{smallstep}(\cdot\langle f,\ lbl:\epsilon \mid r\rangle)$$

where the label field is reset in preparation for the next transition step, which is enabled by the newly introduced prefix dot.

The rewrite rules in $\mathcal{R}^{sos}_{Orc}$ that specify the semantics of the basic Orc expressions are shown in Fig. 6. The rules precisely match the correspondingly labeled SOS rules in Fig. 2.

When executing a site call, according to the site call rewrite rule, the call expression is replaced by the handle expression ($?h$), where h is a fresh handle name generated by a function fresh with respect to the currently used set of handle names η, using a *matching equation*[3] in the condition. Furthermore, the rule emits a message targeted to M into the message pool, adds a site call event label, and updates the handles set. When a site response that corresponds to the call with handle h appears in the message pool, one of the site return rules applies, depending on whether the response is the **stop** value or not. In both cases, the site return rules replace the handle expression with the appropriate Orc expression, add a site return event label, remove the message from the pool, and update the set of handles. The rules for publishing expressions and expression calls are very similar to their counterparts in the SOS specification.

Fig. 7 lists the rewrite rules in $\mathcal{R}^{sos}_{Orc}$ that specify the *synchronous*, instantaneous semantics of the Orc combinators. In the figure, we let $\hat{f}, \hat{g}$ range over active expressions (of the sort AExpr), $\bar{f}, \bar{g}$ over inactive expressions (IExpr), and $\tilde{f}, \tilde{g}$ over non-zero expressions (NZExpr). We also let i denote an internal action label (i.e., a non-site-return label), and n a non-publishing internal action label (i.e., a site call or a τ label). An important distinction between the rewrite rules in $\mathcal{R}^{sos}_{Orc}$ and the SOS rules in Section 3.3, is that the former capture the *synchronous* semantics of Orc expressions whereas the rules in Section 3.3 describe the unrestricted *asynchronous* semantics. This explains the larger number of rules in Fig. 7 compared to those in Fig. 2. Indeed, for each rule in the SOS rules for Orc's combinators in Fig. 2, there are one or more rewrite rules in $\mathcal{R}^{sos}_{Orc}$ that correspond to it. For example, the SOS rule Sym for symmetric parallel composition has two corresponding rewrite rules in $\mathcal{R}^{sos}_{Orc}$, one capturing internal actions for active expressions (SymI), while the other deals with inactive expressions consuming site returns (SymE). Since the symmetric combinator is commutative with identity **0**, the two rewrite rules fully specify the synchronous semantics of parallel composition. Similar observations also apply to the remaining rewrite rules in Fig. 7.

5.2. Orc's synchronous SOS revisited

As described in Section 4.2 and summarized in Fig. 5 above, the distinction between active (non-quiescent) and inactive (quiescent) Orc expressions, which proved useful in obtaining a simple an elegant specification of the instantaneous rewriting semantics of Orc, was captured by defining an order-sorted signature for Orc expressions and values. Since an order-sorted signature Σ with "mix-fix" syntax can be viewed as a context-free grammar G in which the non-terminals of G are the sorts of Σ, and the production rules of G are the mix-fix operator declarations and the subsort declarations in Σ, we may use the synchronous rewriting logic semantics described in Section 5.1 to give a corresponding synchronous SOS of Orc specified by SOS rules that define a single, self-contained transition relation, in which no rule application constraints or further characterizations of transitions are necessary. This connection between order-sorted theories and context-free grammars is exploited by first refining the abstract syntax of Orc given in Fig. 1 to include new syntactic categories NZExpr,

[3] A *matching* equation (see [21, Section 4.3]) of the form $u := v$ is an ordinary equation $u = v$, where u is a constructor pattern with extra variables, say $x_1, \cdots, x_n$, and v is a term. Operationally, it is evaluated by *matching* the canonical form of the substitution instance of v by the equations in the theory modulo the axioms B against the pattern u. In this way, the extra variables become instantiated by matching.

$$\textsc{SymI} : \cdot\langle \hat{f} \mid \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f' \mid \widetilde{g},\ lbl : i \mid r'\rangle \ \text{ if } \ \cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : i \mid r'\rangle$$

$$\textsc{SymE} : \cdot\langle \bar{f} \mid \bar{g},\ lbl : l \mid r\rangle \rightarrow \langle f' \mid \bar{g},\ lbl : h?w \mid r'\rangle \ \text{ if } \ \cdot\langle \bar{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : h?w \mid r'\rangle$$

$$\textsc{Seq1V} : \cdot\langle \hat{f} > x > g,\ lbl : l \mid r\rangle \rightarrow \langle (f' > x > g) \mid [v/x]g,\ lbl : \tau \mid r'\rangle$$
$$\text{if } \cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : !v \mid r'\rangle$$

$$\textsc{Seq1NI} : \cdot\langle \hat{f} > x > g,\ lbl : l \mid r\rangle \rightarrow \langle f' > x > g,\ lbl : n \mid r'\rangle$$
$$\text{if } \cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : n \mid r'\rangle$$

$$\textsc{Seq1NE} : \cdot\langle \bar{f} > x > g,\ lbl : l \mid r\rangle \rightarrow \langle f' > x > g,\ lbl : h?w \mid r'\rangle$$
$$\text{if } \cdot\langle \bar{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : h?w \mid r'\rangle$$

$$\textsc{Asym1V} : \cdot\langle g < x < \hat{f},\ lbl : l \mid r\rangle \rightarrow \langle [v/x]g,\ lbl : \tau \mid r'\rangle$$
$$\text{if } \cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : !v \mid r'\rangle$$

$$\textsc{Asym1NI} : \cdot\langle g < x < \hat{f},\ lbl : l \mid r\rangle \rightarrow \langle g < x < f',\ lbl : n \mid r'\rangle$$
$$\text{if } \cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : n \mid r'\rangle$$

$$\textsc{Asym1NEa} : \cdot\langle \bar{g} < x < \bar{f},\ lbl : l \mid r\rangle \rightarrow \langle \bar{g} < x < f',\ lbl : h?w \mid r'\rangle$$
$$\text{if } \cdot\langle \bar{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : h?w \mid r'\rangle$$

$$\textsc{Asym1NEb} : \cdot\langle \mathbf{0} < x < \bar{f},\ lbl : l \mid r\rangle \rightarrow \langle \mathbf{0} < x < f',\ lbl : h?w \mid r'\rangle$$
$$\text{if } \cdot\langle \bar{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : h?w \mid r'\rangle$$

$$\textsc{Asym2I} : \cdot\langle \hat{g} < x < \widetilde{f},\ lbl : l \mid r\rangle \rightarrow \langle g' < x < \widetilde{f},\ lbl : i \mid r'\rangle$$
$$\text{if } \cdot\langle \hat{g},\ lbl : \epsilon \mid r\rangle \rightarrow \langle g',\ lbl : i \mid r'\rangle$$

$$\textsc{Asym2E} : \cdot\langle \bar{g} < x < \bar{f},\ lbl : l \mid r\rangle \rightarrow \langle g' < x < \bar{f},\ lbl : h?w \mid r'\rangle$$
$$\text{if } \cdot\langle \bar{g},\ lbl : \epsilon \mid r\rangle \rightarrow \langle g',\ lbl : h?w \mid r'\rangle$$

$$\textsc{OtherV} : \cdot\langle \hat{f}\ \mathbf{;}\ \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f',\ lbl : !v \mid r'\rangle \ \text{ if } \cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : !v \mid r'\rangle$$

$$\textsc{OtherNI} : \cdot\langle \hat{f}\ \mathbf{;}\ \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f'\ \mathbf{;}\ \widetilde{g},\ lbl : n \mid r'\rangle \ \text{ if } \cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : n \mid r'\rangle$$

$$\textsc{OtherNE} : \cdot\langle \bar{f}\ \mathbf{;}\ \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f'\ \mathbf{;}\ \widetilde{g},\ lbl : h?w \mid r'\rangle$$
$$\text{if } \cdot\langle \bar{f},\ lbl : \epsilon \mid r\rangle \rightarrow \langle f',\ lbl : h?w \mid r'\rangle$$

Fig. 7. Rewrite rules in $\mathcal{R}^{sos}_{Orc}$ for the combinators.

AExpr and IExpr for the sorts NZExpr, AExpr, IExpr, and refined production rules corresponding to the Orc expression subsort structure (shown in Fig. 5) and the Orc expression operator declarations described in Section 4.2. In particular, in the refined syntax, an Orc expression is either $\mathbf{0}$ or a non-zero expression $\widetilde{f}$, which can be either an active expression $\hat{f}$, or an inactive expression $\bar{f}$. The syntactic categories AExpr and IExpr capture, respectively, the sets of active and inactive expressions, as specified by Definition 1.

Based on this refined syntax, the SOS rules defining the synchronous SOS of Orc are shown in Fig. 8. We note that these rules define a self-contained transition relation specifying the synchronous semantics of Orc, in which no external rule application constraints are necessary. This is in contrast to the approach of [61], in which the transition relation was split into two sub-relations: one for internal transitions on non-quiescent expressions, and another for external transitions on quiescent expressions. We also note that the SOS rules correspond, one-to-one, to the rewriting logic semantics rules of the theory $\mathcal{R}^{sos}_{Orc}$ given in Figs. 6 and 7.

5.3. The tick rule

Following the standard approach for specifying time in rewriting semantic definitions of real-time systems [67] – using either ordinary or real-time rewrite theories, the theory $\mathcal{R}^{sos}_{Orc}$ includes a time *tick* rewrite rule to capture the timed semantics of Orc, in addition to the *instantaneous* rewrite rules in Fig. 6 and Fig. 7. The tick rule in $\mathcal{R}^{sos}_{Orc}$ is a one-step rule defined as follows (with t' of the sort Time):

$$\textsc{Tick} : \cdot\langle f, clk : t \mid r\rangle \rightarrow \langle f', clk : t + t' \mid \delta(r, t')\rangle$$
$$\text{if}\ \mathsf{eager}(\langle f, clk : t \mid r\rangle) \neq \mathsf{true}\ \wedge\ t' := \mathsf{mte}(r)\ \wedge\ t' \neq 0.$$

Note that the variable t' only appears in the righthand side. The value of t' is determined, by the matching equation $t' := \mathsf{mte}(r)$ in the condition, to be the maximum time elapse, which is computed as the minimum message delay across all messages in the message pool of the configuration (see Definition 3). The function δ propagates the effect of a clock tick t' down the record structure of a configuration (somewhat similar to time-shifting in [82]). It essentially updates delays of

$$\frac{h \text{ fresh}}{M(\vec{v}) \xhookrightarrow{M\langle \vec{v}, h\rangle} ?h} \ (\textsc{SiteCall})$$

$$?h \xhookrightarrow{h?v} !v \quad (\textsc{SiteRetV})$$

$$?h \xhookrightarrow{h?\mathbf{stop}} \mathbf{0} \quad (\textsc{SiteRetN})$$

$$!v \xhookrightarrow{!v} \mathbf{0} \quad (\textsc{Publish})$$

$$\frac{E(\vec{x}) \triangleq f \in D}{E(\vec{p}) \xhookrightarrow{\tau} [\vec{p}/\vec{x}]f} \quad (\textsc{Def})$$

$$\frac{\hat{f} \xhookrightarrow{!v} f'}{\hat{f}\,;\, \widetilde{g} \xhookrightarrow{!v} f'} \quad (\textsc{OtherV})$$

$$\frac{\hat{f} \xhookrightarrow{n} f'}{\hat{f}\,;\, \widetilde{g} \xhookrightarrow{n} f'\,;\, \widetilde{g}} \ (\textsc{OtherNI})$$

$$\frac{\bar{f} \xhookrightarrow{r} f'}{\bar{f}\,;\, \widetilde{g} \xhookrightarrow{r} f'\,;\, \widetilde{g}} \ (\textsc{OtherNE})$$

$$\frac{\hat{f} \xhookrightarrow{i} f'}{\hat{f} \mid \widetilde{g} \xhookrightarrow{l} f' \mid \widetilde{g}} \quad (\textsc{SymI})$$

$$\frac{\bar{f} \xhookrightarrow{r} f'}{\bar{f} \mid \bar{g} \xhookrightarrow{r} f' \mid \bar{g}} \quad (\textsc{SymE})$$

$$\frac{\hat{f} \xhookrightarrow{!v} f'}{\hat{f} > x > g \xhookrightarrow{\tau} (f' > x > g) \mid [v/x]g} \ (\textsc{Seq1V})$$

$$\frac{\hat{f} \xhookrightarrow{n} f'}{\hat{f} > x > g \xhookrightarrow{n} f' > x > g} \quad (\textsc{Seq1NI})$$

$$\frac{\bar{f} \xhookrightarrow{r} f'}{\bar{f} > x > g \xhookrightarrow{r} f' > x > g} \quad (\textsc{Seq1NE})$$

$$\frac{\hat{f} \xhookrightarrow{!v} f'}{g < x < \hat{f} \xhookrightarrow{\tau} [v/x]g} \quad (\textsc{Asym1V})$$

$$\frac{\hat{f} \xhookrightarrow{n} f'}{g < x < \hat{f} \xhookrightarrow{n} g < x < f'} \quad (\textsc{Asym1NI})$$

$$\frac{\bar{f} \xhookrightarrow{r} f'}{\bar{g} < x < \bar{f} \xhookrightarrow{r} \bar{g} < x < f'} \ (\textsc{Asym1NEa})$$

$$\frac{\bar{f} \xhookrightarrow{r} f'}{\mathbf{0} < x < \bar{f} \xhookrightarrow{r} \mathbf{0} < x < f'} \ (\textsc{Asym1NEb})$$

$$\frac{\hat{g} \xhookrightarrow{i} g'}{\hat{g} < x < \widetilde{f} \xhookrightarrow{i} g' < x < \widetilde{f}} \quad (\textsc{Asym2I})$$

$$\frac{\bar{g} \xhookrightarrow{r} g'}{\bar{g} < x < \bar{f} \xhookrightarrow{r} g' < x < \bar{f}} \quad (\textsc{Asym2E})$$

Fig. 8. Synchronous (instantaneous) structural operational semantics of Orc.

messages in the message pool, which makes response messages from site calls become eventually available. For example, suppose that the current configuration C is of the form

$$\cdot\langle h_1 \mid h_2, clk : 0 \mid msg : [app(M, v_1, 3), h_1][app(N, v_2, 5), h_2] \mid r\rangle$$

Then, the maximum time elapse of C is 3, and the clock can be advanced by at most $t' = 3$ time units (as otherwise the opportunity of consuming the site return for the call to M would be missed). When the Tick rule is applied, the clock advances to 3, and the function δ updates the delays on all messages in the message pool, arriving at the configuration

$$\cdot\langle h_1 \mid h_2, clk : 3 \mid msg : [app(M, v_1, 0), h_1][app(N, v_1, 2), h_2] \mid r\rangle$$

in which the site return message for M can be processed (as per the definition of *app* on M) and consumed (using the SiteRetV rule).

Ticking the clock, and updating the record structure accordingly, are not enough for the proper timed semantics of Orc, because if not appropriately controlled, new undesirable behaviors may be introduced, such as advancing time indefinitely or beyond a point when an instantaneous action was enabled (and, in effect, *missing* that action). This is avoided by defining a *maximal, time-synchronous* execution semantics, in which an Orc configuration with no enabled instantaneous actions is allowed to advance its clock all the way up to the next point in time when an instantaneous action will be enabled.[4] This restriction is formally specified by making the tick rule conditional on: (1) the configuration *not* being *eager*, i.e., being incapable of making an instantaneous transition as defined by the eager predicate, and (2) the time shift t' being equal to the *maximal time elapse* of the configuration, as defined by the mte function, which must be non-zero (see Section 4.3 for definitions of eagerness and maximal time elapse of configurations). These conditions ensure that time is advanced as much as possible in every application of the tick rule, but only enough so as to be able to capture all events of interest.

5.4. Correctness of $\mathcal{R}^{sos}_{Orc}$

The original SOS transition relation $\hookrightarrow$ proposed in [61] (a variant of which was shown in Fig. 2) and it's refinement, also in [61], into two sub-relations $\hookrightarrow_R$ and $\hookrightarrow_A$ for quiescent and non-quiescent Orc expressions, respectively, defined the synchronous semantics of the instantaneous actions in Orc. Although a non-trivial timed extension of the original SOS specification was later proposed in [82], the extension did not consider the synchronous semantics, and abstracted non-publishing actions as unobservable actions for simplicity of presentation. We, therefore, show correctness of the rewriting

[4] For the analysis to be mechanizable, we also assume Orc programs with “non-Zeno” behaviors [68], such that only a finite number of instantaneous transitions are possible within any finite period of time. In particular, we exclude Orc programs that are not “instantaneously terminating” that may exhibit an infinite sequence of instantaneous transitions in zero time, which in effect can prevent time from ever advancing.

semantics given by $\mathcal{R}^{sos}_{Orc}$ with respect to the SOS semantics in [61] by comparing the transition systems defined by the refined SOS relation $\hookrightarrow_R \cup \hookrightarrow_A$ and the instantaneous part of the rewrite theory $\mathcal{R}^{sos}_{Orc}$, namely, the theory $\mathcal{R}^{sos}_{Orc}$ without the Tick rule.

First, a simple lemma relating the notions of non-quiescent expressions in [61] and active expressions as defined in Section 4.2.

Lemma 1. *f is non-quiescent (resp. quiescent) iff f is active (resp. inactive).*

Proof. Straightforward by structural induction on f. □

We denote by $\mathcal{R}^{sos}_{Orc} \vdash t \rightarrow^1_I t'$ a single rewrite step obtained by an application of an instantaneous (non-tick) rewrite rule, i.e., a rule $I \in \mathcal{R}^{sos}_{Orc} - \{\text{Tick}\}$. Correctness of $\mathcal{R}^{sos}_{Orc}$ is expressed by the following theorem.

Theorem 1 *(Correctness of $\mathcal{R}^{sos}_{Orc}$). For any two Orc expressions f and f', and for $X \in \{A, R\}$,*

$$f \hookrightarrow_X f' \iff \mathcal{R}^{sos}_{Orc} \vdash \mathsf{smallstep}(\cdot\langle f,\ lbl : l \mid r\rangle) \rightarrow^1_I \mathsf{smallstep}(\langle f',\ lbl : l_X \mid r'\rangle)$$

with l_A an internal action label and l_R a site return action label.

Proof. The proof follows trivially by construction of $\mathcal{R}^{sos}_{Orc}$, based on the correctness of the MSOS-to-rewriting logic transformation methodology, given by the strong bisimulation theorem (Theorem 1) in [55], and from Lemma 1. □

5.5. Executability properties of $\mathcal{R}^{sos}_{Orc}$

The specification of the theory $\mathcal{R}^{sos}_{Orc} = (\Sigma^s, E^s \cup B^s, R^s, \phi^s)$ is not only correct with respect to Orc's synchronous semantics but also satisfies some desirable admissibility and executability properties that make it computable and amenable to sound and complete formal analysis and verification. In particular, the signature Σ^s is A^s-preregular [21] and the equations E^s and the rules R^s are deterministic. Furthermore, the equations E^s are operationally terminating, confluent, and sort-decreasing modulo the axioms B^s, and the rules R^s are coherent with E^s. As a result, the specification $\mathcal{R}^{sos}_{Orc}$, through its implementation in Maude as the system module named `SOS-ORC`, can be both executed and formally analyzed. We refer the reader to the extended version of this paper [7] for a detailed proof of the following theorem.

Theorem 2 *(Executability of $\hat{\mathcal{R}}^{sos}_{Orc}$). The specification given by $\hat{\mathcal{R}}^{sos}_{Orc}$ satisfies the executability requirements of generalized rewrite theories.*

6. The reduction rewriting semantics $\mathcal{R}^{red}_{Orc}$

Although the rewriting specification $\mathcal{R}^{sos}_{Orc}$ is readily understandable and its correctness with respect to the SOS semantics in [61] is straightforward, its execution, in practice, is quite expensive and inefficient. This is partly because $\mathcal{R}^{sos}_{Orc}$ makes extensive use of *conditional* rewrite rules (corresponding to the rules in the SOS specifications) which are particularly expensive to execute as compared to unconditional rules. Moreover, most of these rewrite rules, besides being conditional, have rewrites (as opposed to equations) in their conditions, which is typical of the SOS specification style. Rewrite conditions, as opposed to equational conditions, can be particularly expensive to find a proof for or to disprove as they are non-deterministic in nature, and require breadth-first search. In addition, the relatively large number of such rules in the specification can potentially cause nested (recursive) rewrite checks when checking a rule's conditions, which adversely affect performance of execution and analysis.

This section introduces a specification for a rewrite theory $\mathcal{R}^{red}_{Orc}$ that is not directly based on the SOS specifications but is instead more akin to a *reduction semantics*. It utilizes the inherently distributed semantics of rewriting logic, and uses both equations, for modeling *deterministic* computation steps, and rewrite rules, for modeling the *non-deterministic* transitions. This is achieved primarily by localizing the rewrite rules as much as possible, and specifying equationally any required *propagation of information* between the subexpression to be rewritten (the *redex*) and the enclosing Orc configuration (the *context*). In effect, this approach minimizes the number of rewrite rules needed and reduces their complexity, resulting in a semantic specification that can be executed and analyzed much more efficiently. Furthermore, the semantics defined by $\mathcal{R}^{red}_{Orc}$ is equivalent to $\mathcal{R}^{sos}_{Orc}$, in the sense that, given any Orc program P, the state transition systems of the semantics of P given by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ are strongly bisimilar, assuming that program configurations are closed.[5] This implies that $\mathcal{R}^{red}_{Orc}$ captures precisely the intended semantics of Orc while providing an efficient means for execution and formal analysis of Orc programs.

[5] Roughly speaking, a configuration $\langle f, r\rangle$ is *closed* if every expression name referenced in f has a declaration in r (see Section 6.5).

The specification of the reduction rewriting semantics builds on the semantic infrastructure given by $\mathcal{R}_\Omega$ introduced in Section 4, with the main difference that, unlike for $\mathcal{R}^{sos}_{Orc}$, action labels are not essential to $\mathcal{R}^{red}_{Orc}$, since label information is implicitly managed by auxiliary operators in $\mathcal{R}^{red}_{Orc}$. However, to maintain equivalence with $\mathcal{R}^{sos}_{Orc}$, the label field $lbl : l$ is maintained in Orc configurations as before. Furthermore, the one-step modifier strategy used in the SOS-based semantics to implement one-step rewrites is no longer needed.

6.1. The internal actions

Transition steps that correspond to internal actions of Orc expressions are specified using the IAction rewrite rule (with $\mathsf{act}^\uparrow$ an auxiliary function symbol, which will be described shortly):

$$\textsc{IAction} : \langle \hat{f}, r\rangle \to \langle \mathsf{act}^\uparrow(f', i), r\rangle \text{ if } \hat{f} \to \mathsf{act}^\uparrow(f', i).$$

The rule simply states that an (eager) configuration with an active expression may make an internal transition if the expression is able to make that transition. Note that this rule is global at the configuration level, which is required to maintain equivalence with the original interleaving semantics of Orc, and is also essential for executability of the specification. An active expression may make an internal transition according to one of the following rules:

$$\begin{aligned} \textsc{SiteCall} &: M(\vec{v}) \to \mathsf{act}^\uparrow(\mathsf{tmp}, \mathsf{siteCall}(M, \vec{v})) \\ \textsc{ExprCall} &: E(\vec{p}) \to \mathsf{act}^\uparrow(\mathsf{tmp}, \mathsf{exprCall}(E, \vec{p})) \\ \textsc{Publish} &: \ !v \to \mathsf{act}^\uparrow(\mathbf{0}, \mathsf{publish}(v)). \end{aligned}$$

Therefore, an active, basic sub-expression may rewrite to a frozen, auxiliary operator symbol $\mathsf{act}^\uparrow : \mathsf{Expr} \times \mathsf{InternalEvent} \to [\mathsf{Expr}]$, whose purpose is to propagate the action up the expression tree all the way to the top so that: (i) its effects are reflected in the configuration (e.g. emitting a site call message into the configuration), and (ii) any necessary information in the configuration can be propagated back to the sub-expression (e.g. getting globally fresh handle names for site calls). This process of propagating information back and forth between redexes and contexts is specified equationally by induction on the structure of Orc expressions. In particular, for site and expression calls, $\mathsf{act}^\uparrow$ replaces the call with a temporary placeholder expression tmp and propagates the action up to the configuration according to the following equations (where c stands for a site call event $\mathsf{siteCall}(M, \vec{v})$, or an expression call event $\mathsf{exprCall}(E, \vec{p})$):

$$\begin{aligned} \mathsf{act}^\uparrow(f, c) \mid \widetilde{g} &= \mathsf{act}^\uparrow(f \mid \widetilde{g}, c) & \mathsf{act}^\uparrow(f, c) > x > g &= \mathsf{act}^\uparrow(f > x > g, c) \\ \mathsf{act}^\uparrow(f, c) < x < \widetilde{g} &= \mathsf{act}^\uparrow(f < x < \widetilde{g}, c) & \mathsf{act}^\uparrow(f, c) \,;\, \widetilde{g} &= \mathsf{act}^\uparrow(f \,;\, \widetilde{g}, c). \\ g < x < \mathsf{act}^\uparrow(f, c) &= \mathsf{act}^\uparrow(g < x < f, c) \end{aligned}$$

Once the call reaches the root of the expression, the effect of the call is reflected in the containing configuration, using one of the following equations, depending on the call type:

$$\begin{aligned} &\langle \mathsf{act}^\uparrow(f, \mathsf{siteCall}(M, \vec{v})),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle \\ &\quad = \langle \mathsf{act}^\downarrow(f, ?h),\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle \text{ if } h := \mathsf{fresh}(\eta) \end{aligned}$$

$$\begin{aligned} &\langle \mathsf{act}^\uparrow(f, \mathsf{exprCall}(E, \vec{p})),\ lbl : l \mid env : \sigma, E(\vec{x}) \triangleq g \mid r\rangle \\ &\quad = \langle \mathsf{act}^\downarrow(f, [\vec{p}/\vec{x}]g),\ lbl : \epsilon \mid env : \sigma, E(\vec{x}) \triangleq g \mid r\rangle \end{aligned}$$

which capture precisely the semantics of site and expression calls, respectively. The specifications of the effects of site calls and expression calls on the record structure of a configuration are identical to those in the SiteCall and Def rules of the SOS-based semantics of $\mathcal{R}^{sos}_{Orc}$, except that the label field is reset to ϵ. Note that since both the handle h in a site call and the instantiated body g of the expression definition in an expression call need to propagate back to the subterm where the call was made (which was temporarily substituted by the expression tmp), $\mathsf{act}^\uparrow$ does not rewrite immediately to f, but rather to another (frozen) operator, $\mathsf{act}^\downarrow : \mathsf{Expr} \times \mathsf{Expr} \to \mathsf{Expr}$, that traverses down the expression tree until it reaches the appropriate subterm, using the following equations:

$$\begin{aligned} \mathsf{act}^\downarrow(\widetilde{f} \mid \widetilde{f}', g) &= \mathsf{act}^\downarrow(\widetilde{f}, g) \mid \mathsf{act}^\downarrow(\widetilde{f}', g) & \mathsf{act}^\downarrow(\widetilde{f} \,;\, \widetilde{f}', g) &= \mathsf{act}^\downarrow(\widetilde{f}, g) \,;\, \widetilde{f}' \\ \mathsf{act}^\downarrow(\widetilde{f} > x > f', g) &= \mathsf{act}^\downarrow(\widetilde{f}, g) > x > f' & \mathsf{act}^\downarrow(b, g) &= b \\ \mathsf{act}^\downarrow(f < x < \widetilde{f}', g) &= \mathsf{act}^\downarrow(f, g) < x < \mathsf{act}^\downarrow(\widetilde{f}', g) & \mathsf{act}^\downarrow(\mathsf{tmp}, g) &= g, \end{aligned}$$

where b is a basic Orc expression, and g is either a handle expression (for a site call), or the body expression of a declaration (for an expression call).

The action of publishing a value is defined slightly differently, although the overall operational behavior is similar. This is primarily because published values may be bound in an expression by sequential or asymmetric parallel compositions. In

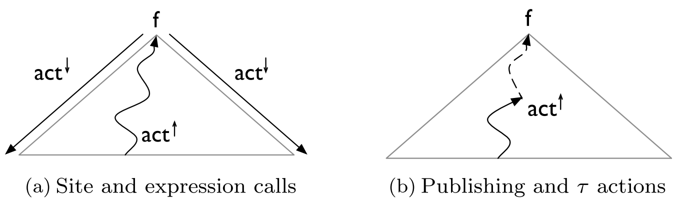


Fig. 9. Schematic diagrams of the equational propagation of information in an expression tree.

particular, if the published value v is *not* bound in the expression, the value is propagated all the way to the top, using the following equations:

$$\begin{aligned} \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) \mid \widetilde{g} &= \mathsf{act}^{\uparrow}(f \mid \widetilde{g}, \mathsf{publish}(v)) \\ \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) < x < \widetilde{g} &= \mathsf{act}^{\uparrow}(f < x < \widetilde{g}, \mathsf{publish}(v)) \\ \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v))\, ;\, \widetilde{g} &= \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)). \end{aligned}$$

In this case, the published value reaches the top of the expression in the enclosing configuration: $\langle \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)), lbl : l \mid r\rangle = \langle f, lbl : \epsilon \mid r\rangle$. Otherwise, if the value published is bound by a sequential composition or an asymmetric parallel composition, then one of the following equations applies:

$$\begin{aligned} &\mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) > x > g = \mathsf{act}^{\uparrow}(f > x > g \mid [v/x]g, \mathsf{publish}^{\tau}) \\ &g < x < \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) = \mathsf{act}^{\uparrow}([v/x]g, \mathsf{publish}^{\tau}). \end{aligned}$$

These equations reflect the semantics specified by the SOS rules Seq1V and Asym1V of Fig. 2 (and the corresponding rewrite rules in $\mathcal{R}^{sos}_{Orc}$). They also change the value publishing event to a τ publishing event *publish*$^{\tau}$, which ultimately causes the label field of the configuration to reset (the equations for terms of the form $\mathsf{act}^{\uparrow}(f, \mathsf{publish}^{\tau})$ are similar). Notice that in both cases, when a publishing (or a τ) event reaches the configuration, no further information needs to be communicated back down the expression, unlike the cases of site and expression calls. Fig. 9 gives a schematic representation of the mechanics of the internal actions. The figure shows that the structures of a site call and an expression call are similar, although the information propagated in both directions (and the side effects on the enclosing configurations) are different.[6]

6.2. The site return action

The external action of a site return is modeled by the following rewrite rule:

$$\begin{aligned} \textsc{SiteReturn} : &\langle \bar{f},\ lbl : l \mid msg : \rho[w,h] \mid hdl : \eta, h \mid r\rangle \\ &\quad \to \langle \mathsf{sret}(\bar{f}, w, h),\ lbl : \epsilon \mid msg : \rho \mid hdl : \eta \mid r\rangle \ \text{ if } h \in \mathsf{handles}(\bar{f}), \end{aligned}$$

which corresponds to the SiteRetV and SiteRetStop rules in the SOS rules and the SOS-based rewrite rules of $\mathcal{R}^{sos}_{Orc}$. Note that application of the site return rule above is subjected to the condition that the handle name of the message to be consumed is referenced in $\bar{f}$. This is to avoid useless transitions that could take place when a thread, having an unfinished site call, is pruned using asymmetric parallel composition, and thus, maintains a comparable behavior to site returns in the SOS specification. In addition, the rule SiteReturn captures the synchronous semantics of Orc by matching an *inactive* expression $\bar{f}$ to consume the site return message. By this rule, the expression $\bar{f}$ rewrites to a frozen auxiliary operator $\mathsf{sret} : \mathsf{Expr} \times \mathsf{ResValue} \times \mathsf{Handle} \to \mathsf{Expr}$ which equationally carries the response parameters down to the appropriate pending handle expression, replacing it with $!w$ if $w \in Value$, or **0** if $w =$ **stop**.

6.3. Timed semantics

Like the SOS-based rewriting semantics $\mathcal{R}^{sos}_{Orc}$, time and the effects of time elapse are specified in the reduction semantics $\mathcal{R}^{red}_{Orc}$ using a simple tick rewrite rule and the δ methodology to propagate the effects of time elapse on the state [67,64]. In fact, the tick rewrite rule is almost identical to that of $\mathcal{R}^{sos}_{Orc}$ given in Section 5.3, but without the SOS one-step modifier. The rule relies on the eager and mte functions to capture the maximal, time-synchronous execution semantics, described before in Section 4.3.

[6] Unwanted concurrent execution of site calls, expression calls and publishing of values is avoided by equations that will introduce an *error* constant of the kind [Expr] in such cases. For reasons of confluence of the equations E^r, an expression having *error* as a subterm immediately collapses to *error* (see [4]).

6.4. *Executability properties of $\mathcal{R}^{red}_{\mathrm{Orc}}$*

Like $\mathcal{R}^{sos}_{\mathrm{Orc}}$, the theory $\mathcal{R}^{red}_{\mathrm{Orc}} = (\Sigma^r, E^r \cup B^r, R^r, \phi^r)$ is executable, which implies that it provides an interpreter for Orc programs, and that formal analysis using its implementation in Maude, as a system module `RED-ORC`, is both sound and complete. A detailed proof of this result, which is stated below as a theorem, is given in [7].

Theorem 3 *(Executability of $\mathcal{R}^{red}_{\mathrm{Orc}}$). The specification given by $\mathcal{R}^{red}_{\mathrm{Orc}}$ satisfies the executability requirements of generalized rewrite theories.*

6.5. *Equivalence of $\mathcal{R}^{sos}_{\mathrm{Orc}}$ and $\mathcal{R}^{red}_{\mathrm{Orc}}$*

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

Definition 4 *(Closed configurations).* An Orc configuration $\langle f, r\rangle$ is *well-formed* if: (i) the configuration $\langle f, r\rangle$ is in canonical form; (ii) f does not contain any auxiliary function symbol, such as $\mathsf{act}^{\uparrow}$, sret, or tmp; and (iii) r contains at least the five fields introduced in Section 4, namely: (1) $lbl : l$, with $l \in \mathsf{Label}$, (2) $hdl : \eta$, with $\eta \in \mathsf{HandleSet}$, (3) $env : \sigma$, with $\sigma \in \mathsf{Env}$, (4) $msg : \rho$, with $\rho \in \mathsf{MsgPool}$, and (5) $clk : t$, with $t \in \mathsf{Time}$. Moreover, a *closed* configuration is a well-formed configuration in which no expression name appears free in f or σ.

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

Lemma 2 *(Preservation of closed configurations). Let $\mathcal{C}$ be a closed configuration. If $\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash \mathcal{C} \to^1 \mathcal{C}'$ for some configuration $\mathcal{C}'$, then $\mathcal{C}'$ is closed. Similarly, if $\mathcal{R}^{red}_{\mathrm{Orc}} \vdash \mathcal{C} \to^1 \mathcal{C}'$, then $\mathcal{C}'$ is closed.*

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

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

The following lemma states that the definitions of eager configurations in $\mathcal{R}^{sos}_{\mathrm{Orc}}$ and $\mathcal{R}^{red}_{\mathrm{Orc}}$ coincide. The lemma is an easy consequence of the fact that the definitions of the eager predicate, the mte function, active and inactive expressions, messages, and auxiliary functions for the time domain and handle names, are all shared in the same semantic infrastructure given by $\mathcal{R}_{\Omega}$.

Lemma 3 *(Timing strategy equivalence). For any configuration $\mathcal{C}$, $\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash \mathsf{eager}(\mathcal{C}) = true$ iff $\mathcal{R}^{red}_{\mathrm{Orc}} \vdash \mathsf{eager}(\mathcal{C}) = true$, and, similarly, $\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash \mathsf{mte}(\mathcal{C}) = t$ iff $\mathcal{R}^{red}_{\mathrm{Orc}} \vdash \mathsf{mte}(\mathcal{C}) = t$, for any $t \in \mathsf{Time}$.*

Now we are ready to present the equivalence theorem, for which a detailed proof can be found in Appendix A.

Theorem 4. *For any configurations $\mathcal{C}$ and $\mathcal{C}'$, the following equivalence holds,*

$$\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash \mathsf{smallstep}(\cdot\mathcal{C}) \to^1 \mathsf{smallstep}(\cdot\mathcal{C}') \iff \mathcal{R}^{red}_{\mathrm{Orc}} \vdash \mathcal{C} \to^1 \mathcal{C}'.$$

The main result of this section can be derived as a consequence of Theorem 4 by taking as $\mathcal{C}$ the *initial configuration* of a program P given by $[P]$ (see Section 4.3).

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

$$\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash [P] \to^1 \mathsf{smallstep}(\cdot\mathcal{C}) \iff \mathcal{R}^{red}_{\mathrm{Orc}} \vdash [P] \to^1 \mathcal{C}.$$

Proof. Immediate from Theorem 4 and the fact that $[P]$ is closed. □

Table 1
A performance comparison of the rewriting semantics of Orc using Maude's `rewrite` and `search` commands (times in milliseconds).

		Timeout	Priority	Par-or	Timed-m	BCast	CList
$\mathcal{R}^{sos}_{Orc}$	rewrite	1.0	1.0	8.0	14.0	3.0	10.0
	search	1.0	2.0	84.0	326.0	5492.0	6.0×10^5
$\mathcal{R}^{red}_{Orc}$	rewrite	1.0	1.0	1.0	1.0	1.0	1.0
	search	1.0	1.0	3.0	4.0	56.0	4.7×10^4

Table 2
A performance comparison of the rewriting semantics of Orc using Maude's LTL model checker applied to four instances of the dining philosophers problem (times in milliseconds).

Problem size	2	3	4	5
$\mathcal{R}^{sos}_{Orc}$	22.0	2423.0	4.8×10^5	∞
$\mathcal{R}^{red}_{Orc}$	3.0	107.0	3230.0	1.6×10^5

6.6. Performance comparison

Despite being bisimilar, the reduction rewriting semantics given by $\mathcal{R}^{red}_{Orc}$ enjoys a significant performance advantage over the SOS-based rewriting semantics given by $\mathcal{R}^{sos}_{Orc}$. This section validates this claim by comparing the formal simulation and analysis performance of the two rewrite theories through their specifications as system modules in Maude.

Throughout all experiments, performance is measured in terms of the CPU time – in milliseconds – taken to perform a particular task as reported by Maude. The tasks are: (1) simulating six Orc programs using Maude's `rewrite` command, (2) exploring the state space of these six programs using Maude's breadth-first `search` command, and (3) model checking deadlock-freedom in four problem instances of a deadlock-free specification of the dining philosophers problem using Maude's LTL model checker. The Orc programs used as benchmarks for these tasks were borrowed from, or inspired by, examples in [61]. In particular, TIMEOUT, which was also given in Section 3.2, PRIORITY, which prioritizes a site call over another, PAR-OR, which specifies a parallel (lazy) disjunction function, and TIMED-M, which makes four timed calls to a site, were all borrowed from [61]. The Orc programs BCast, which implements a *sequential* broadcast, and CList, which constructs *in parallel* a tuple of responses from external sites with timeout, were both inspired by [61]. The expression definitions for BCast and CList are given in Appendix B. For the dining philosophers benchmark, we use the deadlock-free specification given in [61]. For simplicity, we assume no message delays for all the benchmarking tasks above.

The results of these experiments, which were carried out on a 2.93 GHz quad-core machine with 24 GB of memory using Maude 2.6, are summarized in Tables 1 and 2. The results clearly show that the reduction semantics of Orc is much more efficiently executable than the SOS-based semantics, especially when considering complex Orc expressions (having a large number of parallel compositions). The performance gap is more evident when using the `search` command, since searching builds proofs of all reachable states. In addition to having fewer and simpler rewrite rules in $\mathcal{R}^{red}_{Orc}$, attempts to apply the instantaneous action rules in $\mathcal{R}^{red}_{Orc}$ never fail (as can be verified by Maude's profiler) since transition steps corresponding to instantaneous actions are specified only by two rules that match active expressions for internal actions (the rule labeled IAction), and inactive expressions for the external action of site return (the SiteReturn rule). This significantly reduces the need for backtracking-like behaviors when (recursively) searching for proofs of rewrite conditions, which is characteristic of the SOS-based semantics. Finally, the performance advantage of $\mathcal{R}^{red}_{Orc}$ is even more pronounced in the model checking experiments of the notoriously non-deterministic dining philosophers specification, as shown by Table 2. For the SOS-based semantics, the model checker did not finish within a reasonable amount of time for the problem instance with five philosophers.[7]

7. The MOrc tool

MOrc is a web-based formal specification and analysis tool for Orc programs based on Real-Time Maude (RTM) and the real-time rewriting logic specification of Orc, $\mathcal{R}^{red}_{Orc}$.[8] MOrc provides a user-friendly interface for specifying the Orc program or expression to be analyzed, any sites and their definitions, and the desired formal analysis task and its parameters. The tool supports three kinds of formal analyses: (1) simulation, (2) untimed and timed breadth-first search, and (3) untimed and time-bounded model-checking. The tool is designed to balance both simplicity and expressiveness by supporting user inputs in standard Orc notation, hiding interactions with RTM, and providing generic templates for specifying parametric

[7] It is important to note that, like the original SOS specification of Orc, both rewriting semantics are fairly detailed, operational semantics of Orc, and are, as a result, vulnerable to the state space explosion problem, when using the search or LTL model checking commands, particularly for programs with increasing levels of non-determinism.

[8] MOrc is actually based on an object-oriented extension of $\mathcal{R}^{red}_{Orc}$, which is explained in [7].

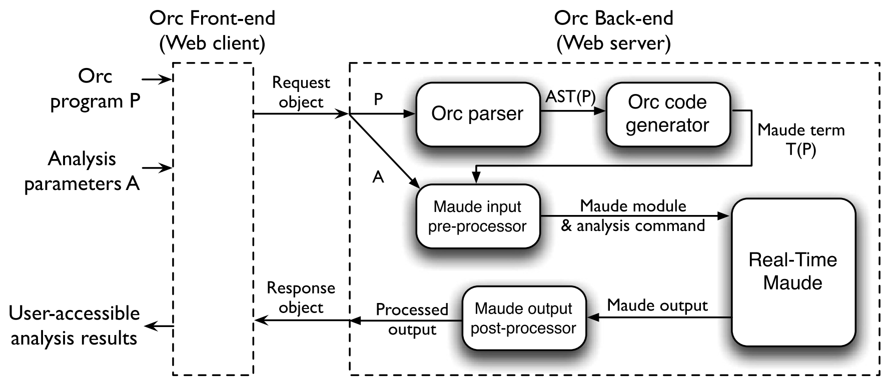


Fig. 10. The architecture of MOrc.

predicates (for both searching and model-checking), using which a wide range of formal properties can be specified. MOrc can be accessed online at http://www.ccse.kfupm.edu.sa/~musab/morc.

7.1. *Components of* MOrc

MOrc is implemented as a dynamic web-based application with both: (1) a front-end client process (implemented using Javascript and the JQuery library) to display appropriate interactive visual elements and manage interactions with the user, and (2) a back-end server (implemented in PHP and C) that pre-processes user input, handles communication with RTM, and post-processes Maude's output. The diagram in Fig. 10 shows the main components of the tool and illustrates their interactions.

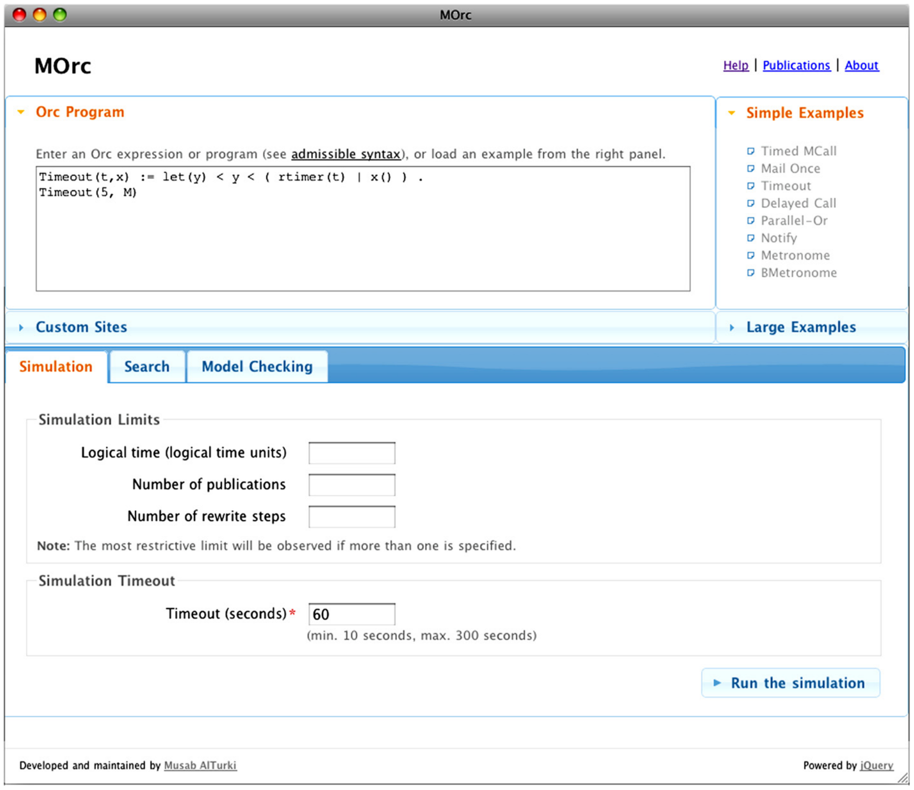
The front-end of MOrc provides an intuitive interface for specifying inputs and displaying analysis results. The main screen, shown in the screen-shot in Fig. 11, is divided into three main sections: (1) an Orc program/expression and site input section, (2) a preloaded examples section, and (3) a tabbed analysis input/output section. The user may wish to load, and then perhaps edit, one of the examples by clicking on it on the right panel, or he/she may specify an entirely different Orc program/expression, essentially using the mathematical notation of the syntax of Orc defined in Section 3.1 (a more precise definition of MOrc-admissible Orc syntax is given in [7]). Furthermore, custom sites can be defined in the "Custom Sites" panel of the program input area by defining the site's name, its parameters, and the values it may publish (only *functional* sites can currently be user-defined). Finally, the formal analysis section presents a tabbed interface with three tabs corresponding to the three analysis modes supported, namely, simulation, search, and model-checking. Each tab provides a customized panel for specifying analysis parameters for the corresponding analysis mode (described below in Section 7.2).

Upon specifying the analysis parameters, the front-end constructs a request object (in Javascript Object Notation format, or JSON) encapsulating all relevant input parameters, submits it to the back-end server, and waits for a response. As illustrated in Fig. 10, the back-end server passes the Orc program text P to a parser and a code generator (both written in C) to build the Maude term $T(P)$ corresponding to the initial state of P. The term $T(P)$, along with the user-supplied analysis parameters, is then fed into an RTM pre-processor that is responsible for generating the appropriate RTM formal analysis command and, for searching and model-checking, a user module that extends the RTM Orc module `ORC` with custom predicates capturing the analysis parameters. After that, the generated module and command are supplied to RTM for execution. The analysis output of RTM is parsed and processed before a server, JSON-encoded object is created and sent to the client. When the client receives the server response object, the front-end replaces the analysis panel in the user interface with a results panel that displays in a structured way the analysis results extracted from the response object.

7.2. *Formal analysis using* MOrc

Simulation. The simulation analysis panel implements RTM's timed rewrite command `trewrite`. The panel may optionally specify limits on the simulation task, which is particularly useful for simulating non-terminating Orc programs. The possible simulation limits are: (1) a *logical time limit*, which specifies an upper bound on the logical time value of the state of the program; (2) a *publications limit*, which specifies an upper bound on the number of publications allowed; and (3) a *rewrite steps limit*, which specifies an upper bound on the number of transitions allowed in the semantics of Orc (where a transition is either a site call, an expression call, the publishing of a value, a site return, or a time tick). If more than one limit is specified, simulation proceeds until at least one of the limits is reached. As an example, we can load the Metronome example, whose declaration was given in Section 3.2, from the "Examples" panel on MOrc's interface:

```
Metronome(t) := let(signal) | rtimer(t) >> Metronome(t) .
Metronome(5)
```



Enter an Orc expression or program (see admissible syntax), or load an example from the right panel.

```
Timeout(t,x) := let(y) < y < ( rtimer(t) | x() ) .
Timeout(5, M)
```

Simulation Limits

Logical time (logical time units)

Number of publications

Number of rewrite steps

Note: The most restrictive limit will be observed if more than one is specified.

Simulation Timeout

Timeout (seconds)* 60

(min. 10 seconds, max. 300 seconds)

Developed and maintained by Musab AlTurki

Powered by jQuery

Fig. 11. The main screen of MOrc's front-end.

Giving a logical time limit of 20 time units, and a publications limit of 2 causes the simulation to stop once the second publication of `signal` is made at logical time 5, with the resulting Orc expression being of the form: `?h » Metronome(5)`. This generally corresponds to the following analysis command in RTM (where `2` is the specified publication limit):

```
(trew {[Metronome(t) := let(signal) | rtimer(t) >> Metronome(t) ;
        Metronome(5), 2]} in time <= 20 .)
```

Finally, a timeout in seconds (which is available in all three analysis panels) must be specified to guard against having simulations running forever. For instance, running the simulation command on METRONOME without specifying any simulation limit causes the tool to display a timeout message.

Search. The search analysis panel implements the breadth-first search command of RTM, which is either timed (using `tsearch`), or untimed (using `utsearch`). Timed search takes into account explicit time-stamps of states, and allows reasoning about timed properties of Orc programs, such as the number of publications within the first t time units, the time at which a specific value is published, and whether a pattern is reachable by a given timeout. When timed search is selected, the user may specify the search time bound, or leave it unspecified for a timed search with no time limit. Untimed search, on the other hand, ignores time-stamps on states, allowing only untimed properties to be specified and checked.

To perform a search, the user typically provides an *Orc expression pattern* that a solution state must satisfy. If specified, the search command will look for states whose Orc expression components match at the top the given pattern. An Orc expression pattern can be either a *concrete* Orc expression, for example `let(x)` and `if(x == y) » let(true)`, or a *symbolic* pattern containing pattern *meta-variables* ranging over terms of appropriate types. Syntactically, a pattern meta-variable is an identifier of the form `$[A-Z][A-Z]?[0-9]*`, where the prefix dollar sign `$` distinguishes pattern meta-variables from Orc variables, and the first one or two uppercase letters specify the type of the terms over which the pattern meta-variable ranges (refer to MOrc's site for supported meta-variables). For example, the pattern `$F > $X > $F` matches a sequential composition of two identical sub-expressions, while the pattern `$M($VL) | $E(1, $P)` matches a parallel composition of an enabled site call and an expression call with two parameters, the first of which is the value 1. Pattern meta-variables, which are internally translated into Maude meta-variables of appropriate sorts, enable symbolic reachability analysis without requiring users of MOrc to be familiar with the underlying Maude syntax.

In addition to Orc expression patterns, semantic constraints on publications of Orc values can also be specified when using search. First, the reachable state space of an Orc program can be constrained by giving an upper bound on the number of publications allowed. Furthermore, it is possible to specify timed constraints (for timed search) or untimed constraints (for untimed search) on what values are published (Type I constraints) or the number of values published (Type II constraints) in a state. MOrc provides two generic templates, corresponding to both types, for the specification of such constraints, which are internally translated into state predicates in RTM. For example, a timed Type I constraint may be that the Orc program must have published a signal within the first five time units of its execution, whereas an untimed Type II constraint may require that the program must have published at least two values. The user may specify as many such constraints as desired through MOrc's interface. For a search task, the solution states will have to satisfy all the specified publication constraints (i.e., the conjunction of the corresponding state predicates in RTM).

As an example, consider the Delayed Response example from the list of examples on the right pane, which is specified as:

```
DelayedResponse(x,t) := rtimer(t) >> let(z) < z < x() .
DelayedResponse(clock, 5) | let(signal)
```

Using timed search with a strict upper bound of 5 time units (i.e. $t < 5$), and using the default values for the other search parameters, we may verify the simple invariant that the site response from `clock` (which is the value 0) is never published before 5 time units have passed. This can be achieved by giving a Type I constraint that looks for states with time-stamps less than 5 in which the value 0 has been published, and making sure that no solution is found. This generally corresponds to issuing the following RTM command (where `pubValue?` is a predicate modeling Type I constraints):

```
(tsearch [10] {[DelayedResponse(x,t) := rtimer(t) >> let(z) < z < x() ;
                DelayedResponse(clock, 5) | let(signal)]}
=>+ {CF:Configuration} such that pubValue?(CF:Configuration, 0, ltr, 5)
in time < 5 .)
```

LTL **Model-checking.** The model checking analysis panel implements the linear temporal logic (LTL) model checking command `mc` of RTM. As for search, model checking is either untimed, so that state time-stamps are abstracted away, or time-bounded, (normally) with a given time bound. The LTL property to be verified in MOrc can be either the generic absence-of-deadlock property, or a custom, user-defined formula, typically based on user-defined atomic predicates. MOrc provides three generic templates for specifying named, parametric atomic predicates: (i) *expression pattern predicate*, which specifies a predicate that is true in a state whose expression component matches at the top the given pattern; (ii) *publication predicate*, which specifies a predicate that is true in a state in which the given value is published within the given time constraints; and (iii) *publication length predicate*, which specifies a predicate that is true in a state in which at least the given number of publications are made within the given time constraints. Furthermore, an upper bound on the number of publications (for time-bounded model-checking) may optionally be specified, which restricts the analysis to the subset of the reachable state space satisfying this bound. The LTL formula field, which is a required field when specifying a custom formula, allows specifying an LTL formula that may make use of user-defined atomic predicates (see [7] for the syntax of LTL formulas).

We illustrate how the model-checking interface panel may be used by means of a specification in Orc of Fischer's protocol with two processes, which is a timed, shared-variable-based, synchronization protocol (assuming atomic reads and writes) for controlling multi-process access to a critical section [43]. The specification in Orc, which can be found in the Examples panel in MOrc by the name FP_{NT}, uses two sites: (1) *shared*, which represents the shared (synchronization) variable, which can be set to a positive value, reset to 0, or waited on (for a possibly unbounded amount of time) to be reset again; and (2) *csection*, which represents the critical section and blocks, when called, for a fixed (finite) period of time before responding. The mutual exclusion property can be verified in FP_{NT} using untimed model checking of the formula: `[] ~ (cs1 /\ cs2)`, where `[]` denotes the LTL "always" operator, `~` the negation operator, `/\` the conjunction operator, and where $\mathtt{cs}_i$, for $i \in \{1, 2\}$, are user-defined atomic predicates specified using pattern expressions of the form `csection(`i`) » $F1 | $F2`. Alternatively, time-bounded model checking (with a reasonable time bound) may also be used to verify this property up to the given bound. Another important property that is desired in such a protocol is absence of *livelock*, so that some progress is being made by either process in the protocol. This can be verified by model checking the formula: `[] <> (cs1 \/ cs2)`, where `<>` denotes the LTL "eventually" operator and `\/` is for disjunction.

8. Related work

Rewriting logic semantics project. Several recent research projects within the *rewriting logic semantics project* [56,73,58], which are closely related to this work, have been conducted. For example, a formal framework for the specification and analysis of timing properties in software design based on RTM was presented at [2]. The framework uses a flexible intermediate language with timeouts to specify software components in a design, and allows for both static analysis (using abstract interpretations) and dynamic analysis (searching and model checking) of timed properties. Another recent example is an RTM-based tool for specifying and analyzing synchronous, real-time embedded software systems in Ptolemy II [9]. The tool implements a code generation infrastructure, similar to MOrc's, minimizing exposure to the underlying Maude model. A third example, also for embedded software components, but with emphasis on safety-critical systems, is given in [65], where an object-based, formal semantics of a behavioral subset of AADL in RTM is used as a formal analysis back-end for AADL specifications. This last work has been extended in [8,10] to handle synchronous AADL models. For a recent survey of these and several other examples, the reader is referred to [54].

Formal semantics of Orc. In addition to the operational SOS semantics of Orc in [61] and its timed SOS extension in [82], several denotational formalizations of Orc's semantics, which are better suited for reasoning about algebraic properties in the language than for describing the operational behavior of Orc programs, have been developed [37,40,69]. The definitive work on Orc is the upcoming [60], which gives a thorough treatment of the Orc semantics, the Orc language, and its powerful features for structured concurrent programming. Encodings of Orc in some other formal models of concurrency, including encodings in the π-calculus [23], Petri nets and the join calculus [17], and networks of timed automata [24, 25], were also given, indirectly providing formal semantics to Orc and highlighting some of its semantic subtleties. The SOS-based rewriting semantics of Orc, along with some of the operational approaches cited above, has similarities with the various SOS semantics that have been given for different timed process calculi, such as ATP [63] and TLP [36], and real-time extensions to various process calculi, such as extensions of ACP [11,12], CCS [20], and CSP [72].

Formal analysis of service compositions. Over the last few years, the problem of formally specifying and verifying service compositions has been approached in several different ways. Below, we selectively highlight some of the most relevant approaches.

An automata-theoretic approach, which was proposed in [24,25] (using a compositional partial order reduction technique to improve performance [76]), leverages available model checking tools for timed automata models, namely Uppaal [13], by modeling the semantics of an Orc expression as a network of timed automata. Unlike MOrc, the resulting Uppaal-based tool enables formal verification of only a subset of Orc with limited recursion, where the number of threads in an Orc expression is fixed. A fundamentally similar approach, but based on an abstraction of BPEL activities instead of Orc, is used in [38], where an abstracted BPEL process is transformed into a network of Web Service Timed Transition Systems (WSTTS), which are essentially timed automata tailored in design for web service compositions. An implementation of the underlying model enables model checking analysis of timed properties, which are specified in discrete-time Duration Calculus. In comparison to the rewriting-based approach of the paper, automata-based methods are limited in expressiveness and are, as a result, insufficient for modeling the full generality of service compositions.

Models of service compositions using Petri nets, and their extensions, have been developed. Most of these approaches, such as [35,84,75], tend to first define a process calculus in which service composition constructs are specified, and give such constructs formal semantics as Petri nets. Correctness of composition specifications can then be verified using standard reachability analysis methods for Petri net models. A fairly recent, introductory book on modeling business processes using Petri nets is also available [77].

Several other (non-Petri-net-based) process-algebraic approaches to service composition specification were developed. The general theme of these methods is to first specify a process calculus with specialized constructs for the targeted aspects of a service composition, like persistent sessions, error handling (exceptions), or security properties, and then develop their formal semantics in some form of a transition system, on which formal verification is based. Examples of such efforts, many of which were partially supported by the Sensoria project [83], include: (1) Service-Centered Calculus (SCC) [14], an Orc-inspired process calculus of service compositions with persistent sessions; (2) Stream-based Service Centered Calculus (SSCC) [44], an SCC-inspired calculus for modeling orchestrations and conversations; (3) Service Oriented Computing Kernel (SOCK) [33], which defines a layered process calculus for modularly specifying service behaviors, service sessions, and service compositions; (4) the Calculus for Orchestration of Web Services (COWS) [45], a timed process calculus with termination constructs; (5) Event Calculus for Web Services [34], a process calculus with events and event scopes for error handling; and (6) Signal Calculus (SC) [29], a variant of the Ambient Calculus [19] for event-notification-based service coordination. Most of these calculi emphasize expressiveness and conciseness by providing constructs that capture different aspects of services, and demonstrate them with examples. It is not clear, however, how mechanizable such formalisms are for performing automatic formal analysis. Bruni [15] provided a survey of such process-calculus-based approaches, and described one of his own, called the Calculus of Sessions and Pipelines (CaSPiS) [15,16], which is a calculus for describing service sessions and their interactions. Further related work on formal analysis of service compositions given in [31] uses game-theoretic methods to formally analyze uncertainty in the behaviors of service orchestrations (expressed in Orc) under bursts of demand.

Other approaches that are based directly on BPEL (and related industrial languages) have also been proposed. Given the fact that BPEL and similar languages are descriptive and verbose, and lack any sort of formal semantics, these approaches essentially try to provide formalizations of (subsets of) these industrial languages in some formal model of computation, like BPEL encodings in Petri Nets [47], the π-calculus [48], Event Calculus [71], and Message Sequence Charts [30]. Alternatively, they may devise new BPEL-inspired *core* languages for service orchestrations that capture some of BPEL's salient features, such as transactions and process termination, including, for example, B*lite* [46], and the BPEL-based process calculi of [79] and [1], focusing on studying correlations between orchestration processes within service choreographies. Nevertheless, as a result of BPEL's complexity and expansive feature-set, a comprehensive and practical BPEL-based formal framework for the specification and verification of service orchestrations remains elusive.

9. Conclusion

Orc [59,61,82] is a very powerful calculus and language for concurrent programming combining great simplicity and mathematical elegance with practical applicability to a wide range of concurrent programming areas. Its semantics is a matter of great importance for theoretical, verification, and implementation purposes. However, Orc's semantics is subtle, due to its real-time aspects and to the distinction between internal and external computation. Faithfully capturing the Orc semantics is therefore a nontrivial challenge for a semantic framework. Furthermore, the real goal should be not just capturing a "paper semantics," but obtaining an executable one from which interpreters, language implementations, and program analysis tools can be derived, essentially for free.

In this work we have shown that the semantic framework of rewriting logic [52] and its support for specification of both concurrency and real-time computation [64] can naturally and elegantly capture all the aspects of the Orc semantics in an executable way. As a side effect, we have also obtained a new formulation of Orcs's synchronous SOS that naturally captures with a single transition relation what required two different transition relations in [61]. For efficiency reasons we have also developed a reduction style semantics strongly bisimilar to the single-step style semantics, but vastly superior in efficiency. This is of great practical importance to make semantics-based Orc tools such as interpreters and model checkers much more efficient.

An important practical result of this work is the development of the MORC semantics-based interpreter and model checker. MORC hides from the user the underlying Real-Time Maude tool where Orc programs are executed and model checked by providing a user interface where only acquaintance with Orc itself is required. This is an instance of a general methodology within the rewriting logic semantics project [51,56,57], where formal methods are hidden "under the hood," so that a language user can perform sophisticated program analyses without having to be familiar with the underlying formalism (see [27,9,8] for other examples of similar Maude-based program analysis tools whose formal underpinnings are hidden "under the hood").

Acknowledgements

We thank Professor Jayadev Misra and all the members of the Orc team at the University of Texas at Austin for explaining to us many details of Orc and its language implementation, and for their very helpful comments on, and encouragement of, the research presented here. We also thank the anonymous reviewers for their valuable comments and suggestions. This work was partially supported by KFUPM Grant JF121005, AFOSR Contract FA8750-11-2-0084 and NSF Grants CCF 09-05584 and CNS 13-19109.

Appendix A. Proof of the equivalence theorem (Theorem 4)

In this section, we refer to the theories $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ respectively by $\mathcal{R}^s$ and $\mathcal{R}^r$, and use $\mathcal{C} \to_{\mathcal{R}} \mathcal{C}'$ to denote $\mathcal{R} \vdash \mathcal{C} \to^1 \mathcal{C}'$, for brevity and notational convenience.

Proof. ($\Longrightarrow$) By induction on a proof of $\mathsf{smallstep}(\cdot\mathcal{C}) \to_{\mathcal{R}^s} \mathsf{smallstep}(\cdot\mathcal{C}')$, which is abbreviated below as $(\cdot\mathcal{C}) \to_{\mathcal{R}^s} (\cdot\mathcal{C}')$. There are six base cases, corresponding to the rules [SITECALL], [PUBLISH], [DEF], [SITERETV], [SITERETSTOP], and [TICK] in $\mathcal{R}^s$:

- [SITECALL]: If h is a fresh handle with respect to a handle set η, and

$$\begin{aligned}&(\cdot\langle M(\vec{v}),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle)\\ &\quad \to_{\mathcal{R}^s} (\langle ?h,\ lbl : M\langle \vec{v}, h\rangle \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle)\\ &\quad =_{\mathcal{R}^s} (\cdot\langle ?h,\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle)\end{aligned}$$

  then, by [IACTION] (and [SITECALL]) in $\mathcal{R}^r$, and using the substitution $\{f' \mapsto \gamma, i \mapsto \mathsf{siteCall}(M, \vec{v})\}$, we have

$$\begin{aligned}&\langle M(\vec{v}),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle\\ &\quad \to_{\mathcal{R}^r} \langle \mathsf{act}^{\uparrow}(\gamma, \mathsf{siteCall}(M, \vec{v})),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle\\ &\quad =_{\mathcal{R}^r} \langle \mathsf{act}^{\downarrow}(\gamma, ?h),\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h]) \mid hdl : \eta, h \mid r\rangle\\ &\quad =_{\mathcal{R}^r} \langle ?h,\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle.\end{aligned}$$

  The base cases for the remaining internal actions, [PUBLISH] and [DEF], are similar.
- [SITERETV]: Suppose

$$\begin{aligned}&(\cdot\langle ?h,\ lbl : l \mid msg : \rho[v, h] \mid hdl : \eta, h \mid r\rangle)\\ &\quad \to_{\mathcal{R}^s} (\langle !v,\ lbl : h?v \mid msg : \rho \mid hdl : \eta \mid r\rangle)\\ &\quad =_{\mathcal{R}^s} (\cdot\langle !v,\ lbl : \epsilon \mid msg : \rho \mid hdl : \eta \mid r\rangle).\end{aligned}$$

Since $?h$ is an inactive expression and $h \in \mathsf{handles}(?h) = \mathit{true}$ is provable from $\mathcal{R}^r$, we use the rule SiteRet in $\mathcal{R}^r$ (with the substitution $\{\bar{f} \mapsto ?h\}$) to get:

$$\begin{aligned}&\langle ?h,\ lbl : l \mid msg : \rho[v,h] \mid hdl : \eta, h \mid r\rangle\\ &\quad \rightarrow_{\mathcal{R}^r} \langle \mathsf{sret}(?h, v, h),\ lbl : \epsilon \mid msg : \rho \mid hdl : \eta \mid r\rangle\\ &\quad =_{\mathcal{R}^r} \langle !v,\ lbl : \epsilon \mid msg : \rho \mid hdl : \eta \mid r\rangle.\end{aligned}$$

The base case for [SiteRetStop] is similar.

- [Tick]: Suppose

$$(\cdot\langle \bar{f}, clk : t \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle \bar{f}, clk : t + t' \mid \delta(r, t')\rangle).$$

Then, $\mathcal{R}^s$ proves $t' = \mathsf{mte}(\mathcal{C})$, which is non-zero, and $\mathsf{eager}(\mathcal{C}) \neq \mathit{true}$. By Lemma 3, $\mathcal{R}^r$, too, proves that t' is the maximum time elapse of $\mathcal{C}$ and that $\mathcal{C}$ is not an eager configuration. Therefore, by the Tick rule in $\mathcal{R}^r$, we have:

$$\langle \bar{f}, clk : t \mid r\rangle \rightarrow_{\mathcal{R}^s} \langle \bar{f}, clk : t + t' \mid \delta(r, t')\rangle.$$

For the inductive step, there are fourteen cases corresponding to the inductive rules listed in Fig. 7. We discuss below representative cases for symmetric parallel and sequential compositions. The remaining cases for asymmetric parallel and otherwise compositions are similar.

- [SymI]. Suppose $(\cdot\langle \hat{f} \mid g,\ lbl : l \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f' \mid g,\ lbl : i \mid r'\rangle)$, which is equationally equivalent to $(\cdot\langle f' \mid g,\ lbl : \epsilon \mid r'\rangle)$, then we have

$$(\cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : i \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f', lbl : \epsilon \mid r'\rangle).$$

By the inductive assumption, this implies $\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f',\ lbl : \epsilon \mid r'\rangle$, by an application of [IAction] in $\mathcal{R}^r$ such that there exists an active base expression as a subexpression of $\hat{f}$. Therefore, $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f'', i)$, with $f'' = \hat{f}[p \leftarrow b]$, for some position p in $\hat{f}$ and b is either tmp or $\mathbf{0}$, depending on whether the internal action is a (site or expression) call or a publishing of a value, respectively. By congruence, this implies $\hat{f} \mid g \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f'', i) \mid g$, which is equal to $\mathsf{act}^{\uparrow}(f'' \mid g, i)$. If the action i is a call, then by the rule [IAction]:

$$\begin{aligned}\langle \hat{f} \mid g,\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r}\ & \langle \mathsf{act}^{\uparrow}(f'' \mid g, i),\ lbl : l \mid r\rangle\\ =_{\mathcal{R}^r}\ & \langle \mathsf{act}^{\downarrow}(f'' \mid g, e),\ lbl : \epsilon \mid r'\rangle\\ =_{\mathcal{R}^r}\ & \langle f' \mid g,\ lbl : \epsilon \mid r'\rangle\end{aligned}$$

where e is a handle expression if i is a site call, or the instantiation of a body of an appropriate expression declaration if i is an expression call. The remaining case when the action i is a publishing action is similar.

- [SymE]. Let u be a site return label, and suppose

$$(\cdot\langle \bar{f} \mid \bar{g},\ lbl : l \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f' \mid \bar{g},\ lbl : u \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f' \mid \bar{g},\ lbl : \epsilon \mid r'\rangle).$$

Then $(\cdot\langle \bar{f},\ lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : u \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f', lbl : \epsilon \mid r'\rangle)$. By the inductive assumption, this implies $\langle \bar{f},\ lbl : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f',\ lbl : \epsilon \mid r'\rangle$, by an application of [SiteReturn] in $\mathcal{R}^r$ such that there exists a handle base expression of the form $?h$ as a subexpression of $\bar{f}$ at some position p, that a message $[w, h]$ exists in the set of handles in the handles field of r, and that f' is either $\bar{f}[p \leftarrow \mathbf{0}]$ or $\bar{f}[p \leftarrow !v]$, depending on whether the return value is a **stop** value or not, respectively. Therefore, by [SiteReturn], we have:

$$\begin{aligned}\langle \bar{f} \mid g,\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r}\ & \langle \mathsf{sret}(\bar{f} \mid g, w, h),\ lbl : \epsilon,\ \mid r'\rangle\\ =_{\mathcal{R}^r}\ & \langle \mathsf{sret}(\bar{f}, w, h) \mid \mathsf{sret}(g, w, h),\ lbl : \epsilon,\ \mid r'\rangle\\ =_{\mathcal{R}^r}\ & \langle f' \mid g,\ lbl : \epsilon,\ \mid r'\rangle.\end{aligned}$$

- [Seq1V]. Suppose

$$(\cdot\langle \hat{f} > x > g,\ lbl : l \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle (f' > x > g) \mid [v/x]g,\ lbl : \tau \mid r'\rangle)$$

which is equationally equivalent to $(\cdot\langle (f' > x > g) \mid [v/x]g,\ lbl : \epsilon \mid r'\rangle)$. Then, $(\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : !v \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f', lbl : \epsilon \mid r'\rangle)$. By the inductive assumption, this implies $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$ by an application of [IAction] in $\mathcal{R}^r$ such that there exists a publishing base expression of the form $!v$ as a subexpression of $\hat{f}$ and that v is not bound in $\hat{f}$. Therefore, $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f', \mathsf{publish}(v))$, with $f' = \hat{f}[p \leftarrow \mathbf{0}]$, for some position p in $\hat{f}$. By congruence, this implies $\hat{f} > x > g \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f', \mathsf{publish}(v)) > x > g$, which is equal to $\mathsf{act}^{\uparrow}(f' > x > g \mid [v/x]g, \mathsf{publish}^{\tau})$, and, thus, by the rule [IAction]:

$$\begin{aligned}\langle \hat{f} > x > g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{act}^{\uparrow}(f' > x > g \mid [v/x]g, \mathsf{publish}^{\tau}),\ lbl : l \mid r\rangle \\ &=_{\mathcal{R}^r} \langle f' > x > g \mid [v/x]g,\ lbl : \epsilon \mid r'\rangle.\end{aligned}$$

- [Seq1NI]. Suppose

$$\begin{aligned}(\cdot\langle \hat{f} > x > g,\ lbl : l \mid r\rangle) &\rightarrow_{\mathcal{R}^s} (\langle f' > x > g,\ lbl : n \mid r'\rangle) \\ &=_{\mathcal{R}^s} (\cdot\langle f' > x > g,\ lbl : \epsilon \mid r'\rangle)\end{aligned}$$

for some internal, non-publishing label n. Then, $(\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : n \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f', lbl : \epsilon \mid r'\rangle)$. By the induction hypothesis, this implies $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$ by an application of [IAction] in $\mathcal{R}^r$ such that $\hat{f}$ has as a subexpression a base expression of one of the following forms: (i) a site call expression $M(\vec{v})$, (ii) an expression call expression $E(\vec{p})$, or (iii) a publishing expression $!v$ with v bound in $\hat{f}$. For case (i), $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f_{tmp}, \mathsf{siteCall}(M, \vec{v}))$, with $f_{tmp} = \hat{f}[p \leftarrow tmp]$, for some position p in $\hat{f}$. By congruence, this implies $\hat{f} > x > g \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f_{tmp}, \mathsf{siteCall}(M, \vec{v})) > x > g$, which is equal to $\mathsf{act}^{\uparrow}(f_{tmp} > x > g, \mathsf{siteCall}(M, \vec{v}))$, and, thus, by the rule [IAction]:

$$\begin{aligned}\langle \hat{f} > x > g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{act}^{\uparrow}(f_{tmp} > x > g, \mathsf{siteCall}(M, \vec{v})),\ lbl : l \mid r\rangle \\ &=_{\mathcal{R}^r} \langle \mathsf{act}^{\downarrow}(f_{tmp} > x > g, ?h),\ lbl : \epsilon \mid r'\rangle \\ &=_{\mathcal{R}^r} \langle f' > x > g,\ lbl : \epsilon \mid r'\rangle.\end{aligned}$$

Cases (ii) and (iii) can similarly be checked using IAction and the appropriate equations in $\mathcal{R}^r$.

- [Seq1NE]. Let u be a site return label, and suppose

$$\begin{aligned}(\cdot\langle \bar{f} > x > g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^s} \langle f' > x > g,\ lbl : u \mid r'\rangle) \\ &=_{\mathcal{R}^s} (\cdot\langle f' > x > g,\ lbl : \epsilon \mid r'\rangle).\end{aligned}$$

Then, $(\cdot\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^s} \langle f', lbl : u \mid r'\rangle)$. By the induction hypothesis, this implies $\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f', lbl : \epsilon \mid r'\rangle$ by an application of [SiteReturn] in $\mathcal{R}^r$ such that there exists a handle base expression of the form $?h$ as a subexpression of $\bar{f}$ at some position p, that a message $[w, h]$ exists in the set of handles in the handles field of r, and that f' is either $\bar{f}[p \leftarrow \mathbf{0}]$ or $\bar{f}[p \leftarrow !v]$, depending on whether the return value is a **stop** value or not, respectively. Therefore, we have by [SiteReturn]:

$$\begin{aligned}\langle \bar{f} > x > g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{sret}(\bar{f} > x > g, w, h),\ lbl : \epsilon,\ \mid r'\rangle \\ &=_{\mathcal{R}^r} \langle \mathsf{sret}(\bar{f}, w, h) > x > g,\ lbl : \epsilon,\ \mid r'\rangle \\ &=_{\mathcal{R}^r} \langle f' > x > g,\ lbl : \epsilon,\ \mid r'\rangle.\end{aligned}$$

The inductive cases for [Asym2I] and [Asym2E] are, respectively, similar to [SymI] and [SymE]. The cases for [Asym1V] and [OtherV] are similar to the value publishing case of [Seq1V], while the cases for [Asym1NI] and [OtherNI] are similar to the internal non-publishing case of [Seq1NI]. Finally, the cases for the external site return action, namely [Asym1NEa], [Asym1NEb] and [OtherNE] are similar to [Seq1NE].

($\Longleftarrow$) If $\mathcal{C} \rightarrow_{\mathcal{R}^r} \mathcal{C}'$ is an instance of the [Tick] rule (i.e., $\mathcal{C}$ is not an eager configuration), then the implication holds trivially by the corresponding [Tick] rule in $\mathcal{R}^s$ by Lemma 3. So, suppose that the rewrite in the hypothesis is not an instance of the tick rule. Then, we observe that it must be an instance of an instantaneous action, which can either be an internal action (with the [IAction] rule) or a site return action (using the [SiteReturn] rule). This implies that the expression component f of $\mathcal{C}$ is either active or inactive (i.e., non-zero). To complete the proof, we proceed by induction on f.

If f is a base active expression (i.e. $M(\vec{v})$, $E(\vec{p})$, or $!v$), then the implication holds easily by the equations in $\mathcal{R}^r$, the assumption that $\mathcal{C}$ is closed, and the corresponding base rules ([SiteCall], [Def], and [Publish]) for these expressions in $\mathcal{R}^s$. Similarly, if f is a base inactive expression, namely the handle expression $?h$, then the implication follows using the corresponding site return rules in $\mathcal{R}^s$ ([SiteRetV] and [SiteRetStop], for the cases of returning an Orc value or a **stop** value, respectively).

Suppose that the expression f is of the form $f \mid g$. If the hypothesis is an instance of an internal action i, then, modulo commutativity, it must be of the form

$$\langle \hat{f} \mid g,\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f' \mid g,\ lbl : \epsilon \mid r'\rangle$$

with $\hat{f}$ having as a sub-expression a base active expression at some position p. Then, $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f'', i)$, with $f'' = \hat{f}[p \leftarrow b]$, for some position p in $\hat{f}$ and b is either *tmp* or $\mathbf{0}$, depending on whether the internal action is a (site or expression) call or a publishing of a value, respectively. By the equations defining $\mathsf{act}^{\uparrow}$ for internal actions, this implies $\langle \hat{f},\ lbl : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f',\ lbl : \epsilon \mid r'\rangle$, which by the induction hypothesis implies $(\cdot\langle \hat{f},\ lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : i \mid r'\rangle)$, and thus the conclusion holds by the rule SYMI in $\mathcal{R}^s$. If the action is an instance of a site return u, then the hypothesis is of the form:

$$\langle \bar{f} \mid \bar{g},\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f' \mid \bar{g},\ lbl : \epsilon \mid r'\rangle$$

with $\bar{f}$ having as a sub-expression a handle expression of the form $?h$, and r having in its messages field an incoming message of the form $[w, h]$, and in its handles field a handle h. By the [SITERETURN] rule and the equations defining sret for site return actions, this implies $\langle \bar{f},\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f',\ lbl : \epsilon \mid r'\rangle$. The induction hypothesis, and the [SYME] rule in $\mathcal{R}^s$ imply the desired conclusion.

Suppose f is of the form $f > x > g$. There are three cases. First, the hypothesis may be an instance of a publishing action, and hence of the form

$$\langle \hat{f} > x > g,\ lbl : l \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle(f' > x > g) \mid [v/x]g,\ lbl : \epsilon \mid r'\rangle$$

with $\hat{f}$ having as a sub-expression a base publishing expression $!v$ such that v is not bound in $\hat{f}$. By the [IACTION] and [PUBLISH] rules, and the equations defining $\mathsf{act}^{\uparrow}$ for publishing actions, this implies $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$. By induction, and rule [SEQ1V] in $\mathcal{R}^s$, the conclusion holds.

The second case is when the action is a *non-publishing* internal action (i.e. a site call or a τ action). In this case the hypothesis has the form:

$$\langle \hat{f} > x > g,\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f' > x > g,\ lbl : \epsilon \mid r'\rangle$$

where, by the [IACTION] rule, $\hat{f}$ has as a sub-expression either a site call, an expression call or a publishing expression in a value-binding position. Again, [IACTION] and the equations defining $\mathsf{act}^{\uparrow}$ for non-publishing internal actions, imply $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$. Induction and rule [SEQ1NI] in $\mathcal{R}^s$ complete the proof of this case

Finally, the action may be an external site return action, and the hypothesis has the form:

$$\langle \bar{f} > x > g,\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f' > x > g,\ lbl : \epsilon \mid r'\rangle$$

which, by the [SITERETURN] rule, the equations defining *sret*, the inductive hypothesis and the $\mathcal{R}^s$ rule [SEQ1NE], implies the desired conclusion.

The inductive cases for the asymmetric parallel composition and the otherwise composition are similar and follow by induction and the corresponding rules in $\mathcal{R}^s$. □

Appendix B. BCAST and CLIST expression declarations

The Orc program BCAST has the following declaration:

$$\begin{aligned} BCast(m, x) \triangleq (&if(empty(x)) \gg let(\mathsf{signal}) \\ &\mid if(\neg empty(x)) \gg head(x) > a > a(m) \gg BCast(m, tail(x))) \end{aligned}$$

where *empty*, *head*, and *tail* are (local) sites implementing list functions.

The program CLIST has the following declaration:

$$\begin{aligned} CList(x) \triangleq\ &(if(empty(x)) \gg let([]) \\ &\mid if(\neg empty(x)) \gg head(x) > a > (append(y, ys) \\ &\quad < y < (a() \mid Rtimer(5) \gg let(\mathsf{signal})) \\ &\quad < ys < CList(tail(x)))) \end{aligned}$$

with *append* a local site that appends an element to a list.

References

[1] Faisal Abouzaid, John Mullins, Formal specification of correlation in WS orchestrations using BP-calculus, in: Proceedings of the 5th International Workshop on Formal Aspects of Component Software, FACS 2008, in: Electron. Notes Theor. Comput. Sci., vol. 260, 2010, pp. 3–24.

[2] Musab AlTurki, Dinakar Dhurjati, Dachuan Yu, Ajay Chander, Hiroshi Inamura, Formal specification and analysis of timing properties in software systems, in: Marsha Chechik, Martin Wirsing (Eds.), Fundamental Approaches to Software Engineering, in: Lect. Notes Comput. Sci., vol. 5503, Springer, 2009, pp. 262–277.

[3] Musab AlTurki, José Meseguer, Real-time rewriting semantics of Orc, in: PPDP '07: Proceedings of the 9th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming, ACM Press, New York, NY, USA, 2007, pp. 131–142.

[4] Musab AlTurki, José Meseguer, Rewriting logic semantics of Orc, Technical Report UIUCDCS-R-2007-2918, University of Illinois at Urbana Champaign, November 2007, http://hdl.handle.net/2142/11410.
[5] Musab AlTurki, José Meseguer, Reduction semantics and formal analysis of Orc programs, Electron. Notes Theor. Comput. Sci. 200 (3) (2008) 25–41.
[6] Musab AlTurki, José Meseguer, Dist-Orc: a rewriting-based distributed implementation of Orc with formal analysis, in: The 1st International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS), Longyearbyen, Spitsbergen, Norway, April 2010.
[7] Musab AlTurki, José Meseguer, Executable rewriting logic semantics of Orc and formal analysis of Orc programs, Technical report, University of Illinois at Urbana Champaign, September 2012, http://hdl.handle.net/TBD.
[8] Kyungmin Bae, Peter Csaba Ölveczky, Abdullah Al-Nayeem, José Meseguer, Synchronous AADL and its formal analysis in Real-Time Maude, in: Shengchao Qin, Zongyan Qiu (Eds.), Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM) 2011, in: Lect. Notes Comput. Sci., vol. 6991, Springer, 2011, pp. 651–667.
[9] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, Edward A. Lee, Stavros Tripakis, Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude, Sci. Comput. Program. 77 (12) (2012) 1235–1271.
[10] Kyungmin Bae, Peter Csaba Ölveczky, José Meseguer, Abdullah Al-Nayeem, The SynchAADL2Maude tool, in: Juan de Lara, Andrea Zisman (Eds.), FASE, in: Lect. Notes Comput. Sci., vol. 7212, Springer, 2012, pp. 59–62.
[11] J.C.M. Baeten, J.A. Bergstra, Real time process algebra, Form. Asp. Comput. 3 (2) (1991) 142–188.
[12] J.C.M. Baeten, Cornelis A. Middelburg, Process Algebra with Timing, Monographs in Theoretical Computer Science, Springer, Berlin, New York, 2002.
[13] G. Behrmann, A. David, K.G. Larsen, A tutorial on UPPAAL, in: Marco Bernardo, Flavio Corradini (Eds.), International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT), in: Lect. Notes Comput. Sci., vol. 3185, Springer, 2004, pp. 200–236.
[14] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, Scc: a service centered calculus, Lect. Notes Comput. Sci. 4184 (2006) 38.
[15] Roberto Bruni, Calculi for service-oriented computing, in: Proceedings of SFM'09, in: Lect. Notes Comput. Sci., vol. 5569, Springer, 2009, pp. 1–41.
[16] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, Ugo Montanari, Hierarchical models for service-oriented systems, in: Martin Wirsing, Matthias M. Hölzl (Eds.), Results of the SENSORIA Project, in: Lect. Notes Comput. Sci., vol. 6582, Springer, 2011, pp. 349–368.
[17] Roberto Bruni, Hernán Melgratti, Emilio Tuosto, Translating Orc features into Petri nets and the Join calculus, in: Mario Bravetti, Manuel Núñez, Gianluigi Zavattaro (Eds.), Web Services and Formal Methods, in: Lect. Notes Comput. Sci., vol. 4184, Springer, 2006, pp. 123–137.
[18] Roberto Bruni, José Meseguer, Semantic foundations for generalized rewrite theories, Theor. Comput. Sci. 360 (1–3) (2006) 386–414.
[19] Luca Cardelli, Andrew D. Gordon, Mobile ambients, Theor. Comput. Sci. 240 (1) (2000) 177–213.
[20] Liang Chen, An interleaving model for real-time systems, in: TVER '92: Proceedings of the Second International Symposium on Logical Foundations of Computer Science, Springer-Verlag, London, UK, 1992, pp. 81–92.
[21] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Carolyn Talcott, All About Maude – A High-Performance Logical Framework, Lect. Notes Comput. Sci., vol. 4350, Springer-Verlag, Secaucus, NJ, USA, 2007.
[22] William Cook, Sourabh Patwardhan, Jayadev Misra, Workflow patterns in Orc, in: Paolo Ciancarini, Herbert Wiklicky (Eds.), Coordination Models and Languages, in: Lect. Notes Comput. Sci., vol. 4038, Springer, Berlin/Heidelberg, 2006, pp. 82–96.
[23] William R. Cook, Jayadev Misra, A structured orchestration language, http://www.cs.utexas.edu/users/wcook/Drafts/OrcCookMisra05.pdf, July 2005.
[24] Jin Song Dong, Yang Liu, Jun Sun, Xian Zhang, Verification of computation orchestration via timed automata, in: Zhiming Liu, Jifeng He (Eds.), Formal Methods and Software Engineering, Proceedings of the 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1–3, 2006, in: Lect. Notes Comput. Sci., vol. 4260, Springer, 2006, pp. 226–245.
[25] JinSong Dong, Yang Liu, Jun Sun, Xian Zhang, Towards verification of computation orchestration, Form. Asp. Comput. 26 (4) (2014) 729–759.
[26] Chucky Ellison, Grigore Rosu, An executable formal semantics of C with applications, in: John Field, Michael Hicks (Eds.), Proceedings of the 39th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012, ACM, 2012, pp. 533–544.
[27] A. Farzan, F. Cheng, J. Meseguer, G. Roşu, Formal analysis of Java programs in JavaFAN, in: Proc. CAV'04, in: Lect. Notes Comput. Sci., vol. 3114, 2004.
[28] Azadeh Farzan, José Meseguer, Grigore Roşu, Formal JVM code analysis in JavaFAN, in: Charles Rattray, Savitri Maharaj, Carron Shankland (Eds.), Algebraic Methodology and Software Technology, in: Lect. Notes Comput. Sci., vol. 3116, Springer, Berlin, Heidelberg, 2004, pp. 132–147.
[29] Luigi Gian Ferrari, Roberto Guanciale, Daniele Strollo, JSCL: a middleware for service coordination, in: Proceedings of FORTE'06, in: Lect. Notes Comput. Sci., vol. 4229, Springer, 2006, pp. 46–60.
[30] H. Foster, S. Uchitel, J. Magee, J. Kramer, Model-based verification of web service compositions, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, October 2003, pp. 152–161.
[31] Joaquim Gabarro, Maria Serna, Alan Stewart, Analysing web-orchestrations under stress using uncertainty profiles, Comput. J. 57 (11) (2014) 1591–1615, http://dx.doi.org/10.1093/comjnl/bxt063.
[32] Joseph Goguen, José Meseguer, Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations, Theor. Comput. Sci. 105 (1992) 217–273.
[33] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, G. Zavattaro, Sock: a calculus for service oriented computing, Lect. Notes Comput. Sci. 4294 (2006) 327.
[34] C. Guidi, R. Lucchi, M. Mazzara, A formal framework for web services coordination, Electron. Notes Theor. Comput. Sci. 180 (2) (2007) 55–70.
[35] R. Hamadi, B. Benatallah, A Petri net-based model for web service composition, in: Proceedings of the 14th Australasian Database Conference, vol. 17, 2003, pp. 191–200.
[36] Matthew Hennessy, Tim Regan, A process algebra for timed systems, Inf. Comput. 117 (2) (1995) 221–239.
[37] Tony Hoare, Galen Menzel, Jayadev Misra, A tree semantics of an orchestration language, in: Engineering Theories of Software Intensive Systems, 2005, pp. 331–350.
[38] Raman Kazhamiakin, Paritosh Pandya, Marco Pistore, Timed modelling and analysis in web service compositions, in: ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security, IEEE Computer Society, Washington, DC, USA, 2006, pp. 840–846.
[39] D. Kitchin, E. Powell, J. Misra, Simulation using orchestration, in: Proceedings of AMAST, January 2008.
[40] David Kitchin, William R. Cook, Jayadev Misra, A language for task orchestration and its semantic properties, in: CONCUR 2006, in: Lect. Notes Comput. Sci., vol. 4137, Springer, 2006, pp. 477–491.
[41] David Kitchin, Adrian Quark, William Cook, Jayadev Misra, The Orc programming language, in: David Lee, Antónia Lopes, Arnd Poetzsch-Heffter (Eds.), Formal Techniques for Distributed Systems, Proc. of FMOODS/FORTE, in: Lect. Notes Comput. Sci., vol. 5522, Springer, 2009, pp. 1–25.
[42] David Kitchin, Adrian Quark, Jayadev Misra, Quicksort: combining concurrency, recursion, and mutable data structures, in: A.W. Roscoe, Cliff B. Jones, Kenneth R. Wood (Eds.), Reflections on the Work of C.A.R. Hoare, History of Computing, Springer, London, 2010, pp. 229–254.
[43] Leslie Lamport, A fast mutual exclusion algorithm, ACM Trans. Comput. Syst. 5 (January 1987) 1–11.
[44] Ivan Lanese, Francisco Martins, Vasco T. Vasconcelos, Antonio Ravara, Disciplining orchestration and conversation in service-oriented computing, in: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, SEFM '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 305–314.
[45] A. Lapadula, R. Pugliese, F. Tiezzi, Cows: a timed service-oriented calculus, Lect. Notes Comput. Sci. 4711 (2007) 275.
[46] A. Lapadula, R. Pugliese, F. Tiezzi, A formal account of WS-BPEL, Lect. Notes Comput. Sci. 5052 (2008) 199.

[47] Niels Lohmann, A feature-complete Petri net semantics for WS-BPEL 2.0, in: Proceedings of WS-FM'07, in: Lect. Notes Comput. Sci., vol. 4937, Springer, 2008, pp. 77–91.
[48] R. Lucchi, M. Mazzara, A pi-calculus based semantics for WS-BPEL, J. Log. Algebr. Program. 70 (1) (2007) 96–118.
[49] Patrick Meredith, Mark Hills, Grigore Roşu, An executable rewriting logic semantics of K-scheme, in: Danny Dube (Ed.), Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Laval University, 2007, pp. 91–103, Technical Report DIUL-RT-0701.
[50] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, Grigore Roşu, A formal executable semantics of Verilog, in: Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10), IEEE, 2010, pp. 179–188.
[51] J. Meseguer, G. Roşu, Rewriting logic semantics: from language specifications to formal analysis tools, in: Proc. Intl. Joint Conf. on Automated Reasoning, IJCAR'04, Cork, Ireland, July 2004, in: Lect. Notes Comput. Sci., vol. 3097, Springer, 2004, pp. 1–44.
[52] José Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155.
[53] José Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce (Ed.), Proc. WADT'97, in: Lect. Notes Comput. Sci., vol. 1376, Springer, 1998, pp. 18–61.
[54] José Meseguer, Twenty years of rewriting logic, J. Log. Algebr. Program. 81 (7–8) (2012) 721–781.
[55] José Meseguer, Christiano Braga, Modular rewriting semantics of programming languages, in: Algebraic Methodology and Software Technology, 2004, pp. 364–378.
[56] José Meseguer, Grigore Rosu, The rewriting logic semantics project, Theor. Comput. Sci. 373 (3) (2007) 213–237.
[57] José Meseguer, Grigore Rosu, The rewriting logic semantics project: a progress report, in: Olaf Owe, Martin Steffen, JanArne Telle (Eds.), Fundamentals of Computation Theory, in: Lect. Notes Comput. Sci., vol. 6914, Springer, 2011, pp. 1–37.
[58] José Meseguer, Grigore Roşu, The rewriting logic semantics project: a progress report, Inf. Comput. 231 (1) (2013) 38–69.
[59] Jayadev Misra, Computation orchestration: a basis for wide-area computing, in: Manfred Broy (Ed.), Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, in: NATO ASI Ser., 2004.
[60] Jayadev Misra, Structured concurrent programming, Manuscript, University of Texas at Austin, January 2013.
[61] Jayadev Misra, William R. Cook, Computation orchestration: a basis for wide-area computing, J. Softw. Syst. Model. 6 (1) (March 2007) 83–110.
[62] Peter D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60–61 (2004) 195–228.
[63] X. Nicollin, J. Sifakis, The algebra of timed processes ATP: theory and application, Inf. Comput. 114 (1) (1994) 131–178.
[64] P.C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, High.-Order Symb. Comput. 20 (1–2) (2007) 161–196.
[65] Peter Ölveczky, Artur Boronat, José Meseguer, Formal semantics and analysis of behavioral AADL models in Real-Time Maude, in: John Hatcliff, Elena Zucca (Eds.), Formal Techniques for Distributed Systems, in: Lect. Notes Comput. Sci., vol. 6117, Springer, Berlin/Heidelberg, 2010, pp. 47–62.
[66] Peter Csaba Ölveczky, Real-Time Maude 2.3 manual, http://heim.ifi.uio.no/~peterol/RealTimeMaude/, August 2007.
[67] Peter Csaba Ölveczky, José Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theor. Comput. Sci. 285 (August 2002) 359–405.
[68] Peter Csaba Ölveczky, José Meseguer, Abstraction and completeness for Real-Time Maude, Electron. Notes Theor. Comput. Sci. 176 (4) (2007) 5–27.
[69] Sidney Rosario, David Kitchin, Albert Benveniste, William Cook, Stefan Haar, Claude Jard, Event structure semantics of Orc, in: WS-FM 2007, in: Lect. Notes Comput. Sci., vol. 4937, Springer, 2008, pp. 154–168.
[70] Grigore Rosu, Andrei Stefanescu, Matching logic: a new program verification approach, in: Richard N. Taylor, Harald Gall, Nenad Medvidovic (Eds.), Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011, ACM, 2011, pp. 868–871.
[71] M. Rouached, O. Perrin, C. Godart, Towards formal verification of web service composition, Lect. Notes Comput. Sci. 4102 (2006) 257.
[72] Steve Schneider, Jim Davies, D.M. Jackson, George M. Reed, Joy N. Reed, A.W. Roscoe, Timed CSP: theory and practice, in: Proceedings of the Real-Time: Theory in Practice, REX Workshop, Springer-Verlag, London, UK, 1992, pp. 640–675.
[73] Traian Florin Serbanuta, Grigore Rosu, José Meseguer, A rewriting logic approach to operational semantics, in: Special Issue on Structural Operational Semantics (SOS), Inf. Comput. 207 (2) (2009) 305–340.
[74] Mark-Oliver Stehr, CINNI — a generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi, in: Kokichi Futatsugi (Ed.), Proceedings of the Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000, in: Electron. Notes Theor. Comput. Sci., vol. 36, Elsevier, 2000, pp. 71–92.
[75] Jindian Su, Shanshan Yu, Heqing Guo, Formal description and verification of web service composition based on OOPN, in: De-Shuang Huang, Donald Wunsch, Daniel Levine, Kang-Hyun Jo (Eds.), Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues, in: Lect. Notes Comput. Sci., vol. 5226, Springer, Berlin/Heidelberg, 2008, pp. 644–652.
[76] Tian Huat Tan, Yang Liu, Jun Sun, Jin Song Dong, Verification of orchestration systems using compositional partial order reduction, in: Shengchao Qin, Zongyan Qiu (Eds.), ICFEM, in: Lect. Notes Comput. Sci., vol. 6991, Springer, 2011, pp. 98–114.
[77] Wil van der Aalst, Christian Stahl, Modeling Business Processes: A Petri Net Oriented Approach, MIT Press, Cambridge, MA, 2011.
[78] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, Bartek Kiepuszewski, Alistair P. Barros, Workflow patterns, Distrib. Parallel Databases 14 (1) (2003) 5–51.
[79] Mirko Viroli, Towards a formal foundation to orchestration languages, in: Proceedings of the First International Workshop on Web Services and Formal Methods, WSFM 2004, in: Electron. Notes Theor. Comput. Sci., vol. 105, 2004, pp. 51–71.
[80] Patrick Viry, Equational rules for rewriting logic, Theor. Comput. Sci. 285 (2) (2002) 487–517.
[81] I. Wehrman, D. Kitchin, W. Cook, J. Misra, Properties of the timed operational and denotational semantics of Orc, Technical report, University of Texas at Austin, 2007.
[82] Ian Wehrman, David Kitchin, William R. Cook, Jayadev Misra, A timed semantics of Orc, Theor. Comput. Sci. 402 (2–3) (2008) 234–248.
[83] Martin Wirsing, Rocco De Nicola, Stephen Gilmore, Matthias Hölzl, Roberto Lucchi, Mirco Tribastone, Gianlugi Zavattaro, Sensoria process calculi for service-oriented computing, in: Proceedings of TGC'06, in: Lect. Notes Comput. Sci., vol. 4661, 2007, pp. 30–50.
[84] Wen Zhao, Yu Huang, Chongyi Yuan, Lifu Wang, Formalizing business process execution language based on Petri nets, in: 2010 2nd International Workshop on Intelligent Systems and Applications (ISA), 2010, pp. 1–8, 22–23.