Verification of Computation Orchestration Via Timed Automata

Jin Song Dong, Yang Liu[⋆], Jun Sun, and Xian Zhang

School of Computing,
National University of Singapore
Tel.: +65 68742834; Fax: +65 6779 4580
{dongjs, liuyang, sunj, zhangxi5}@comp.nus.edu.sg

Abstract. Recently, a promising programming model called *Orc* has been proposed to support a structured way of orchestrating distributed web services. *Orc* is intuitive because it offers concise constructors to manage concurrent communication, time-outs, priorities, failure of sites or communication and so forth. The semantics of *Orc* is also precisely defined. However, there is no verification tool available to verify critical properties against *Orc* models. Instead of building one from scratch, we believe the existing mature model-checkers can be reused. In this work, we first define a Timed Automata semantics for the *Orc* language, which we prove is semantically equivalent to the original operational semantics of *Orc*. Consequently, Timed Automata models are systematically constructed from *Orc* models. The practical implication of the construction is that tool supports for Timed Automata, e.g., UPPAAL, can be used to model check *Orc* models. An experimental tool is implemented to automate our approach.

1 Introduction

The prevalence of the Internet and web services raises the request of service-oriented computing [22], which can invoke remote services, process the results and communicate results with other terminals. However, it is very difficult and complex to design an orchestrating system with concurrency and synchronization using practical programming languages because these traditional languages use threads for concurrency and semaphores for synchronization. Even the higher-level libraries, like channel and working pool, have to be built up based on these primary elements.

Recently, a promising programming language Orc [17, 6] has been proposed for orchestrating distributed services in a structured manner. It abstracts all computations, web services and time control mechanisms as site calls, which are implemented by primitive remote procedures. With this abstraction, it provides a concise syntax for concurrent site call executions, threads synchronization and message passing. In addition, slow response and service failure can be easily handled using timing site calls. Using Orc, complicated orchestrating problems can be easily understood and constructed without worrying about the programming details.

Orc is as well precise and elegant. Both operational semantics [17] and denotational semantics (a tree semantics [18]) are defined. However, as a new emerging language,

[⋆] Corresponding author.

Z. Liu and J. He (Eds.): ICFEM 2006, LNCS 4260, pp. 226–245, 2006.

there are no formal verification mechanisms to systematically verify critical properties over systems modelled in Orc. In this work[1], we address the verification problem of the Orc language. Our aim is to detect possible violations of critical properties, especially timing properties, of Orc programs using an existing mature model checker. Our approach starts with defining an executable model in Timed Automata [1] for Orc expressions, which conforms with the semantics of the Orc language as defined in [6]. As a natural consequence, existing tool support for Timed Automata, e.g., UPPAAL [4], can be used for verification of Orc models. We use two examples, namely Auction Site and Purchase Order Handling System, to demonstrate our approach. Moreover, we implement a tool to construct UPPAAL models automatically from Orc models. Constrained by the Timed Automata theories, our approach focuses a subset of Orc language that is regular, type-safe and with a finite number of threads.

Orc has a strong theoretical foundation in process algebras, particularly CCS [15], CSP [12] and π-calculus [16]. These process algebras provide fundamental models of concurrency in which processes communicate over channels. However, Orc is different from the above process algebras as Orc permits integration of arbitrary components (sites) in a computation. More importantly, Orc has timing control to handle the site failures. Traditional process algebras have well established model checking theories and tool supports, e.g., FDR2 [20] for CSP, and FO$\lambda^{\Delta\nabla}$ [24] for π-calculus. Because of the absence of quantitative timing support, none of these tools can model and verify timing aspects of complex systems. There are some process algebras with time extensions, e.g., Timed CSP [21]. Unfortunately, there is no good model checker available[2]. Timed Automaton [1] is a notation developed for modelling and verification of real-time systems. It is a specialized finite state machine with clocks. Well developed automatic verification tools are available for Timed Automata [4, 7, 23]. This gives the inspiration of this work. Our Timed Automata semantics for Orc would allow Timed Automata verification techniques, theories and tools, to be applied to Orc.

Our work is related to works on BPEL4WS verification [10, 19] as BPEL4WS shares many common elements with Orc. BPEL4WS [13] (Business process orchestration languages for web services) is an XML based business process orchestration language. Both BPEL4WS and Orc orchestrate the web services by using process composition (sequential and parallel) and communication (synchronous and asynchronous). However they are different in several ways. BPEL4WS has a rich set of the language structures to ease the process design. Orc's concise syntax allows the reuse of the process definitions. BPEL4WS has variables to store the state of the communications and is able to receive calls from client web services. Orc is more abstract and as it focuses on process and communication. Orc has a well-defined semantics. Our work therefore focuses on defining an equivalent semantics for Orc in Timed Automata so as to use existing tools.

The rest of the paper is organized as follows: Section 2 briefly introduces the Orc language and the notation of Timed Automata. Section 3 presents an executable modelling

[1] Besides this work, our research team recently starts to work on a reasoning tool for Timed CSP.

[2] To our knowledge, the only tool support for TCSP is the preliminary PVS encoding of TCSP in Brooke's PhD thesis [5].

in Timed Automata for each and every constructor in Orc. Section 4 demonstrates how UPPAAL is used to verify the Orc language using two case studies. Section 5 concludes the paper with possible future works.

2 Background

This section is devoted to a brief introduction to the relevant languages and notations, namely the Orc computation model and Timed Automata.

2.1 Orchestration Language Orc

The syntax and informal semantics of Orc are described in this section. Formal definition of Orc semantics can be found elsewhere at [6].

In the following syntax, E is an expression name, M a site name, x a variable, c a constant, P a list of actual parameters and Q a list of formal parameters.

$$\begin{array}{lll} D \in \mathit{Decl} & ::= & E(Q) \mathrel{\widehat{=}} f \\ f, g \in \mathit{Expression} & ::= & \mathbf{0} \parallel M(P) \parallel E(P) \parallel f >x> g \parallel f \mid g \parallel f \text{ }\mathbf{where}\text{ } x :\in g \\ p \in \mathit{Actual} & ::= & x \parallel c \parallel M \\ q \in \mathit{Formal} & ::= & x \parallel M \end{array}$$

Declaration $E(Q) \mathrel{\widehat{=}} f$ defines expression E whose formal parameter list is Q and body is expression f. An expression is either elementary or a composition of two expressions. An elementary expression is either: (1) **0**, a site which never responds, (2) a site call $M(P)$, or (3) an expression call $E(P)$. Orc has three composition operators: (1) $>x>$ for sequential composition, (2) $|$ for symmetric parallel composition, and (3) **where** for asymmetric parallel composition.

Site. The basic element of Orc expression is a site call. A site is a separately defined procedure, e.g., a web service implemented on a remote machine. A site call can give at most one response; it is possible that a site never responds to a call, which is treated as non-terminating computation. A site call has the same form as a function call: the name of a site followed by an optional list of parameters. For example, calling site $\mathit{Google}(w)$ where Google is an internet search engine and w is a keyword, may return the web sites links related to the keyword. Calling $\mathit{Email}(a, m)$ sends message m to address a, causing a permanent change in the recipient's mailbox, and returns a signal to denote completion of the operation. Site calls are strict, i.e., a site is called only if all its parameters have values. Table 1 lists the fundamental sites used in Orc for effective programming.

Sequential Composition Operator. Sequential operator $>x>$ allows strict sequencing of site calls. For example, $\mathit{Google}(w) > m > \mathit{Email}(a, m)$ will first call site Google, and name the returned value as m. After that $\mathit{Email}(a, m)$ is called. If either site fails to respond, then the evaluation returns no value. The simpler notation $M \gg N$ is used when the value returned by site M is of no significance. To send two emails in sequence and then call Notify, we write

$$\mathit{Email}(\mathit{addr}1, m) \gg \mathit{Email}(\mathit{addr}2, m) \gg \mathit{Notify}$$

Table 1. Fundamental Sites

0	never responds. It can be used to terminate a computation.
$let(x, y, ...)$	returns a tuple consisting of the values of its arguments.
Clock	returns the current time at the server of this site as an integer.
$Atimer(t)$	where t is integer and $t \geq Clock$, returns a signal at time t.
$Rtimer(t)$	where t is integer and $t \geq 0$, returns a signal after exactly t time units.
$if(b)$	where b is boolean, returns a signal if b is true, and remains silent (no response) if false.
Signal	returns a signal immediately. It is the same as $Rtimer(0)$.

Symmetric Parallel Operator. Symmetric parallel operator | gives the power of multi-threaded computation. Evaluation of $f \mid g$, creates two threads to compute f and g respectively. The result from $f \mid g$ is the interleaving of these two streams in time order. If both threads produce values simultaneously, they are merged arbitrarily. Operator | is commutative and associative. An interesting expression is $(Google(w) \mid Yahoo(w)) > m > Email(a, m)$. Here, the first part $(Google(w) \mid Yahoo(w))$ may publish multiple values, and for each value v, we call $Email(a, m)$ where m is set to v. Therefore, the evaluation can cause up to two emails to be sent, one with the value from *Google* and the other from *Yahoo*.

Asymmetric Parallel Operator. The asymmetric parallel operator **where** is used to prune portions of a computation selectively: $Email(a, m)$ **where** $m :\in (Google(w) \mid Yahoo(w))$ sends at most one email, with the first value received from either *Google* or *Yahoo*. In this expression, $Email(a, m)$ and $(Google(w) \mid Yahoo(w))$ are evaluated simultaneously. $Email(a, m)$ is blocked because m does not have a value. Evaluation of $(Google(w) \mid Yahoo(w))$ may return up to two values; the first value is assigned to m and further evaluation of this expression is then terminated. After that, $Email(a, m)$ is unblocked and executed.

Expression Definition. An expression is defined like a procedure, with a name and possible parameters, though it may return a stream of values. As an example, consider the following restaurant reservation process, where $R1$ and $R2$ are two restaurants, and t is the meal time. The user is notified for the first acknowledgement received from the two restaurants, if any.

$$Reservation(t) \mathrel{\widehat{=}} Notify(x) \textbf{ where } x :\in R1(t) \mid R2(t)$$

Recursive definition is also supported in Orc. The following expression defines a *Clock* using $Rtimer(t)$, which emits a signal every time unit, starting immediately.

$$Clock \mathrel{\widehat{=}} Signal \mid Rtimer(1) \gg Clock$$

Dining Philosophers. An example of using Orc is the classical dining philosophers problem, originally presented in [17]. There are N Philosophers, sitting around a table. Every pair of neighbors shares a fork. The fork to the left of Philosopher i is $Fork_i$ and to his right is $Fork_{i'}$ ($i' = (i+1)$ *mod* N). Philosopher i can eat only if it holds both left and right forks. A philosopher's life cycle consists of the following activities: acquire the two adjacent forks, eat, and release the forks. Because of the seating arrangement, neighboring philosophers can not eat simultaneously.

Each $Fork_i$ is modelled as a FIFO buffered channel which is either empty (if some philosopher holds the corresponding fork) or has one signal (if no philosopher holds the fork). We write $Fork_i.put$ to send a signal along the channel and $Fork_i.get$ to get a signal from the channel. Initially, each channel holds a signal. In this example, P_i $(0 \leq i < N)$ depicts philosopher i, where the right neighbor of P_i is $P_{i'}$ (i' = $(i+1)$ *mod* N), and *Eat* returns a signal on completion of eating.

$$\begin{aligned} P_i \mathrel{\widehat{=}} (&let(x,y) \gg Eat \gg Fork_i.put \gg Fork_{i'}.put \\ &\textbf{where } x :\in Fork_i.get, y :\in Fork_{i'}.get) \gg P_i \end{aligned}$$

The dining philosophers problem can be represented as:

$$DP \mathrel{\widehat{=}} P_0 \mid P_1 \mid \cdots \mid P_{N-1}$$

This definition of dining philosophers can lead to deadlock. To avoid deadlock, philosophers should pick up their forks in a specific order. For instance, all except P_0 pick up their left and then their right forks, and P_0 picks up its right and then its left fork.

$$\begin{aligned} &P'_0 \mathrel{\widehat{=}} Fork_1.get \gg Fork_0.get \gg Eat \gg Fork_1.put \gg Fork_0.put \gg P'_0 \\ &P'_i(1 \leq i < N) \mathrel{\widehat{=}} Fork_i.get \gg Fork_{i'}.get \gg Eat \gg Fork_i.put \gg Fork_{i'}.put \gg P'_i \\ &DP' \mathrel{\widehat{=}} P'_0 \mid P'_1 \mid \cdots \mid P'_{N-1} \end{aligned}$$

2.2 Timed Automata and UPPAAL

Timed Automata are finite state machines equipped with clocks. It is a formal notation to model behaviors of real-time systems. Its definition provides a general way to annotate state transition graphs with timing constraints using finitely many real-valued clock variables. Given a set of clock C, the set of clock constraints $\Phi(C)$ is defined as:

$$\phi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \phi_1 \wedge \phi_2$$

where x is a clock variable and c is a real number.

Definition 1 (Timed Automata). *A timed automaton $\mathcal{A}$ is a 6-tuple $\langle S, s_0, \Sigma, C, I, T\rangle$, where S is a finite set of states, s_0 is the initial state, Σ is the alphabet, C is a finite set of clocks, $I : S \rightarrow \Phi(C)$ is a mapping from a state to a state invariant, and $T \subseteq S \times \Sigma \times 2^C \times \Phi(C) \times S$ is the transition relation.* □

In Timed Automata, a state is associated with an invariant, while a transition is labelled with a synchronization action, a guard (a constraint on clocks) and a clock reset (a set of clocks to be reset). Intuitively, a timed automaton starts execution with all clocks initialized to zero. The automaton can stay at a node, as long as the invariant of the node is satisfied, with all clocks increasing at the same rate. A transition can be taken if the values of the clocks fulfill the guard. By taking the transition, all clocks in the clock reset are set to zero, while the clocks not in the clock reset keep their values. For example, Figure 1 illustrates some simple timed automata. Graphically, a double-lined circle indicates an initial state. Typically, a Timed Automata modelling of complex systems would consist of a network of timed automata[3].

Definition 2 (Timed Automata Network). *A network of timed automata is the parallel composition of a collection of $\mathcal{A}_1, \ldots, \mathcal{A}_n$, denoted as $\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$. A transition of the network of timed automata is either a local step of one of the automata where $(s_1, e, c, i, s_2) \in \mathcal{A}_i \wedge e \notin (\bigcup_{k:1..n \wedge k \neq i} \Sigma_k)$ or a pairwise synchronization between two automata where $(s_1, e!, c, i, s_2) \in \mathcal{A}_i$ and $(s_1', e?, c', i', s_2') \in \mathcal{A}_j$.* □

UPPAAL [4] is our choice of model-checker for verifying a network of timed automata because of its efficiency (both for model-checking and simulation) as well as its wide recognition. UPPAAL is a tool for modelling, simulation and verification of real-time systems modelled as timed automata. It consists of three main parts, a system editor which provides a graphical interface to design timed automata, a simulator and a model checker. The simulator is a validation tool which enables examination of possible dynamic executions of a system and thus provides an inexpensive mean of fault detection prior to verification by the model checker which covers the exhaustive dynamic behavior of the system. The model checker checks invariant and bounded liveness properties by exploring the symbolic state space of a system. The properties are expressed as a rich subset of TCTL [11]. In a nutshell, UPPAAL is a model checker for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real valued clocks, communicating through channels or shared variables. Typical applications include real-time controllers and communication protocols, e.g., those where timing aspects are critical. In this work, we extend its application to orchestration of web services.

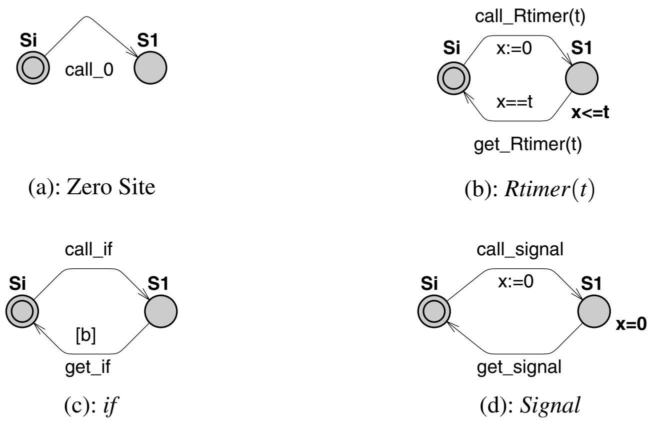
3 Timed Automata Semantics for Orc

This section is devoted to a definition of Timed Automata semantics for Orc models, which allows us to systematically construct the Timed Automata model from an Orc model. The practical implication is that we may then reuse existing tools and theories for Timed Automata to achieve various purposes, for instance, synthesis of implementation [3], simulation [2], theorem proving [14] or more importantly formal verification [4]. In the following, the Timed Automata semantics for Orc expressions is formally defined. The dining philosophers example is used as a running example.

Definition 3 (Zero Site). *A zero site* $\mathbf{0}$ *is modelled as an automaton $\mathcal{A}_{\mathbf{0}}$ where $S = \{s_i, s_1\}$ and $\Sigma = \{call_{\mathbf{0}}\}$ and $C = \varnothing$ and $I = \varnothing$ and $T = \{s_i, call_{\mathbf{0}}, \varnothing, true, s_1\}$.* □

A zero site is a site that never responds. Thus there is no *publish* event, as illustrated in Figure 1(a). The formal definition of the automaton for the fundamental site $Rtimer(t)$ is presented below, which plays the central role in the timing aspect of the orchestration.

[3] We may treat an automata network as one automata by constructing the product. However, leaving it as a network saves us from the state space explosion problem as well as allowing us to benefit from optimization built in the timed automata tools.



(a): Zero Site (b): *Rtimer*(t) (c): *if* (d): *Signal*

Fig. 1. Fundamental Sites

Definition 4 ($Rtimer(t)$)**.** *A* $Rtimer(t)$ *site is modelled as an automaton* $\mathcal{A}_{Rtimer(t)}$ *where* $S = \{s_i, s_1\}$ *and* $\Sigma = \{call_{Rtimer(t)}, get_{Rtimer(t)}\}$ *and* $C = \{x\}$ *and* $I = \varnothing$ *and* $T = \{(s_i, call_{Rtimer(t)}, \{x\}, true, s_1), (s_1, get_{Rtimer(t)}, \varnothing, x = t, s_i)\}$. □

The Timed Automaton for $Rtimer(t)$ is illustrated in Figure 1(b). Once the site is called via the synchronization on the $call_{Rtimer(t)}$ event, the local clock x is reset to 0. After exactly t time units, the calling site is notified via the $get_{Rtimer(t)}$ event. Notice that we adopt the synchronous semantics of Orc in this definition. In the asynchronous semantics, arbitrary delays in processing events are allowed, including the $call_{Rtimer(t)}$ event. Consequently, all we can assert about the call to $Rtimer(t)$ is that client will receive the signal *sometime* after t unit delay, which is too weak for program time-outs or timed-interrupts. We believe that the synchronous semantics is intuitive and powerful. However, the asynchronous semantics can be easily captured by changing the $\Phi(C)$ on the transition from s_1 to s_i as $x \geq t$ and removing the state invariant on state s_1.

Similarly, fundamental sites $call_{if}$ and $call_{Signal}$ are defined as timed automata as well, which are illustrated in Figure 1 (c) and (d) respectively. $call_{Atimer(t)}$ is ignored since $Atimer(t)$ can represented as $Rtimer(t - c)$, where c is the current clock value. $call_{let}$ is a simple Timed Automaton similar to $call_{if}$, but the second transition is the *publish* event without condition b.

The fundamental sites presented so far are defined as the complete expression calls (see definition 9). If we only consider timed automata for the Orc contracts of the fundamental sites, then the *call* events should be removed, e.g., the zero site **0** contains just a single state without any transitions.

Definition 5 (Site Call). *A site call* $M(P)$ *is modelled as an automaton* $\mathcal{A}_{M(P)}$ *where* $S = \{s_i, s_1, s_2, s_3\}$, $\Sigma = \{call_{M(P)}, get_{M(P)}, publish_{M(P)}\}$, $C = \varnothing$, $I = \varnothing$, *and* $T = \{(s_i, call_{M(P)}, \varnothing, true, s_1), (s_1, get_{M(P)}, \varnothing, true, s_2), (s_2, publish_{M(P)}, \varnothing, true, s_3)\}$. □

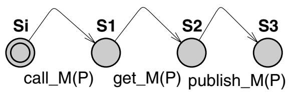


Fig. 2. TA for Site Call

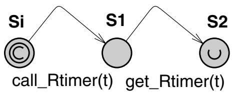


Fig. 3. TA for *Rtimer*(t) Call

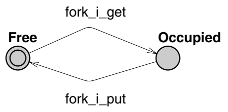


Fig. 4. TA for *Fork*$_i$

A site call is modelled as a timed automaton allowing a *call* event which invokes the service and a *get* event which gets the response from the called site and a *publish* event which publishes the response, illustrated in Figure 2. This conforms the operational semantics of site call, i.e., the three steps of invocation, response, publication as in [6].

A special kind of site calls is the calls to *Rtimer*(t) and *Signal* because of the timing constraints. The invocation of *Rtimer*(t) site is shown in Figure 3 (*Signal* calls are ignored for the similarity). The initial state is set as committed state[4], which will fire the outgoing event $call_{Rtimer(t)}$ immediately with the top priority among all transitions. The finishing state is set as an urgent state[5], which stops the timer in the finishing state. By using the committed and urgent states, we can get exactly t time units between the initial state and finishing state.

The behavior of the external called site must be specified as a separate timed automaton for the sake of verification. For example, the behaviors of the forks in the dining philosophers example are modelled as in Figure 4, where the user may repeatedly get the fork and then put it back. Consequently, a site call *Fork*$_i$*.put* is interpreted as a synchronization on the $call_{Fork_i.out}$ (simplified as *Fork*$_i$*.put* in this example). For an abstract site call like *Eat*, instead of building a trivial automaton which synchronizes on the *call* event and then returns a signal, it is treated as an abstract local event for the sake of efficient verification[6].

Definition 6 (Sequential Composition). *Let the automata network of g be* $\mathcal{A}_g \mathrel{\widehat{=}} \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$. *A sequential composition f >x> g is modelled as a timed automata network* $\mathcal{A}_{f>x>g} \mathrel{\widehat{=}} \mathcal{A}_f \parallel \mathcal{A}'_g$ *where,* $\mathcal{A}'_g \mathrel{\widehat{=}} (\mathcal{A}'_1 \parallel \ldots \parallel \mathcal{A}'_n)^k$ *and for all* $i : 1 \ldots n$, $\mathcal{A}'_i \mathrel{\widehat{=}} \langle S, s_i, \Sigma, C, I, T\rangle$ *where* $S = \mathcal{A}_i.S \cup \{s_i\}$ *and* $\Sigma = \mathcal{A}_i.\Sigma \cup \{publish_x\}$ *and* $C = \mathcal{A}_i.C$ *and* $I = \mathcal{A}_i.I$ *and* $T = \mathcal{A}_i.T \cup \{(s_i, publish_x, \varnothing, true, \mathcal{A}_i.s_i)\}$. □

Notice that a channel[7] named $publish_x$ is defined to synchronize the publishing of a value of x and the receiving of the value.

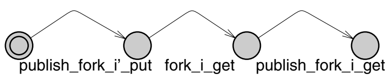
A sequential composition is modelled as, in general, a network of timed automata. The network of f is untouched, whereas the automata in the network of g have to synchronize on the event $publish_x$ before making a step. If there is no value passing between

[4] In UPPAAL, committed states freeze time. If any process is in a committed location, the next transition must involve an edge from one of the committed locations.

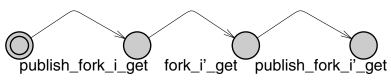
[5] In UPPAAL, urgent states are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location. Hence, time is not allowed to pass when the system is in an urgent location.

[6] In UPPAAL, it corresponds to a transition labelled with no channel event.

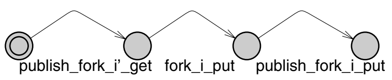
[7] In UPPAAL, a broadcast channel is used here in order to do the synchronization for all paralleled automata in the g.



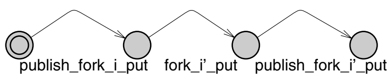
(a) TA for $Fork_i.get$



(b) TA for $Fork_{i'}.get$



(c) TA for $Fork_i.put$



(d) TA for $Fork_{i'}.put$

Fig. 5. Network of Automata for $P'_i(1 \leq i \leq N)$

the Orc expressions, the first publishing signal, i.e., event $publish_x$, is used to precede the automata for expression g.

To abuse the notations, we use $\mathcal{A}^k$ to denote a network containing k copies of the same automaton $\mathcal{A}$. The network of f is parallel-composed with multiple copies of network of g. Every time a new value of x is published, a new instance of the g component is created and starts execution. In general, there would be infinite number of overlapping activations of the g component. However, if we assume the g part executes reasonably fast (and terminating), we need only a finite number of copies of g to fork and reuse them once they are terminated. For the sake of verification of real world applications, we always assume that there is an upper bound on the number of overlapping activation of the g part. For example, Figure 5 presents the automata interpretation of the $P'_i(1 \leq i \leq N)$ in the dining philosophers example, where each site call is model as a TA and local event *eat* has been removed for simplicity. In general, multiple copies of each of the automata is required. However, only one copy for each automaton is shown as that is all that is needed in this case.

Definition 7 (Symmetric Parallel Composition). *A symmetric parallel composition $f \mid g$ is modelled as a network of two timed automata (networks) $\mathcal{A}_f \parallel \mathcal{A}_g$.* □

A symmetric parallel composition is modelled as two automata (networks) running in parallel. There is no communication between the f and g. f and g are probably remote site call to services which run independently on remote machines. Thus, two automata (networks) sharing no common event are used to capture the interleaving behaviors. For example, the automata network for DP' in the dining philosophers example is the network containing the networks in Figure 5 (one for each i).

The last compositional constructor of Orc is the asymmetric parallel composition, denoted f **where** $x :\in g$. According to the semantics in [6], the g expression terminates as soon as one value of x is published. This kind of dynamic termination of timed automata is achieved through the use of a shared global flag.

Definition 8 (Asymmetric Parallel Composition). *Let flag be a global boolean variable. It is initially true. Let the network of the expression g be $\mathcal{A}_g \mathrel{\widehat=} \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$. An asymmetric parallel composition f* **where** *$x :\in g$ is modelled as a network of timed automata $\mathcal{A}_{f\ \mathbf{where}\ x:\in g} \mathrel{\widehat=} \mathcal{A}_f \parallel \mathcal{A}'_g$ where, $\mathcal{A}'_g = \mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_n$ and for all $i : 1 \mathinner{..} n$, $\mathcal{A}'_i \mathrel{\widehat=} \langle \mathcal{A}_i.S, \mathcal{A}_i.s_i, \mathcal{A}_i.\Sigma, \mathcal{A}_i.C, \mathcal{A}_i.I, T \rangle$ where*

$T = \{(s_1, publish_x, cl', gc, s_2) \mid (s_1, publish_x, cl, gc, s_2) \in \mathcal{A}_i.T\}$
$\quad \cup\{(s_1, e, cl, gc \wedge flag, s_2) \mid e \neq publish_x \wedge (s_1, e, cl, gc, s_2) \in \mathcal{A}_i.T\}$
*wherecl'setsflagtofalseandresetstheclocksinclusingassignmentin*UPPAAL. □

As soon as a publishing of x is achieved, the global flag is set to be *false* (this is atomic since they are on the same transition). Consequently all transitions in the network of the expression g are blocked. Therefore, the network of g terminates. Notice that the flag is carefully implemented so that it is local to the automata in $\mathcal{A}'_g$ (by defining a unique global variable for each activation of the network). The execution of $\mathcal{A}_f$ is not blocked until a synchronization on event $publish_x$ is required. Therefore, it may make steps in parallel or even before g does. We remark that while our definitions of timed automata interpretation for Orc expression are generic, there are plenty of simplifications and optimizations to be performed on the constructed timed automata. For example, the P_i expression is modelled (and simplified) as the automaton in Figure 6.

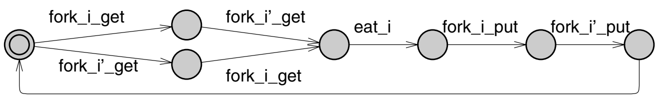


Fig. 6. Timed Automata for P_i

Definition 9 (Expression Call). *An expression call is $E(P)$ with $E(P) \widehat{=} f$ is modelled as the network of timed automata for f prefixed by the $call_E(P)$ event, i.e., $\mathcal{A}_{E(P)} \widehat{=} \langle S, s_i, \Sigma, C, I, T\rangle$, where $S = \{s_i \cup \mathcal{A}_f.S\}$ and $\Sigma = \{call_{E(P)} \cup \mathcal{A}_f.\Sigma\}$ and $C = \mathcal{A}_f.C$ and $I = \mathcal{A}_f.I$ and $T = \{(s_i, call_{E(P)}, \varnothing, true, \mathcal{A}_f.s_i) \cup \mathcal{A}_f.T\}$. .* □

For each parameter x of the expression call, a channel $publish_x$ is defined to synchronize with the publishing of a value of the parameter x. In case there are multiple parameters, the expression call is executed only after all the parameters get their values (via synchronization on the corresponding channels). Publishing of the parameters may occur in any order.

For simple recursion where there is only one automaton instead of an automata network when the recursion call is reached (with our simplification and optimization done), we connect the last state to the initial state to make a loop, e.g., the automaton in Figure 6. In general, recursion is resolved by replacing it with the least fixed point. However, Orc does allow expressions like $N = f \mid N$ where there could be infinite number of copies of f. These kinds of expressions are disallowed for the sake of model checking.

The soundness of the Timed Automata modelling is proved by showing that there is a weak bi-simulation relation between the timed automata and the operational semantics of Orc. The following theorem is proved by a structural induction over our definitions and the operational semantics of Orc defined in [17] (see Appendix for the proof details).

Theorem 1. *For any Orc expression*[8] *f, $\mathcal{A}_f \approx \mathcal{O}_f$, where $\mathcal{O}_f$ is the state transition system constructed from the operational semantics of Orc in [17].* □

[8] We focus on a subset of Orc langauge that is regular, type-safe and with a finite number of threads (see Section 4 for details).

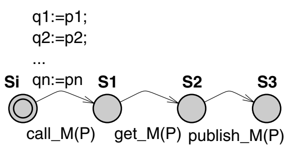


Fig. 7. TA for Site Call with Value Passing

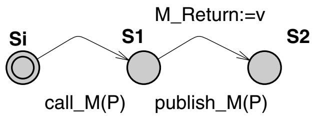


Fig. 8. TA for Site with Value Passing

Value Passing Handling
Timed Automata do not have notions for variables and assignments. Fortunately UPPAAL as an extension of Timed Automata introduces variables (both local and global) and variable assignments (in events). Hence parameter passing can be realized through some globally shared variables since no data can be attached along a channel communication. It is obvious that these shared variables must have unique names. Because the names of site calls are unique, we prefix all the formal parameters' names with their site call names. The return values of each site call are named as site call name + “Return”.

To invoke a site call, the formal parameters are assigned to the value of actual parameters in the *call* event in the Site Call model. The complete model of $call_{M(P)}$ is shown in Figure 7. The return value of a site is assigned in the *publish* event in the Site model (Figure 8). The sequential composition $f\ {>}x{>}\ g$ has an additional assignment $x := f_{Return}$ for variable x in the *publish* event of f. Similarly for asymmetric parallel composition f **where** $x :\in g$, we add the assignment $x := g_{Return}$ in the *publish* event of g. The expression call $E(P) \mathrel{\widehat{=}} f$ has also an assignment in the *publish* event for its return value.

4 Verification Using UPPAAL

This section is devoted to a discuss on how to use tool support for Timed Automata, in particular UPPAAL, to formally analyze the constructed Timed Automata. In general, our modelling of Orc may end up with a network containing an infinite number of automata (see Definitions 6 and 9). One piece of evidence of an possibly infinite number of automata is that Orc in general allows an irregular language (as in automata theory). Our target is therefore a subset of Orc langauge that is regular, type-safe and only allows a finite number of threads. Some Orc examples that we regard as problematic are as the following:

$P \mathrel{\widehat{=}} b \mid a \gg P \gg c,$ where a, b, c are sites or even expressions
$M \mathrel{\widehat{=}} f(x)\ where\ let(0) \mid Signal$
$N \mathrel{\widehat{=}} x$ **where** $x :\in N$

P in general allows the language of the form $a^n bc^n$ which is a typical example of an irregular language. It is a known fact that such languages can not be expressed using finite automata. Therefore, they are beyond automata-based model checking. M is not type safe because the type of x can be either integer 0 or a signal. In general, x could be any type. This as well presents a problem to current model-checking techniques. Lastly, *N* allows an infinitely number of threads of *f* running independently, which would result in an infinite internal loop without returning a value, i.e., a divergence in CSP's terms.

4.1 Automated Construction

We developed an experimental tool to automatically construct UPPAAL models from Orc models using XML and Java technology. We start with parsing the Orc program and building an Abstract Syntax Tree. Afterwards, each Orc language construct is converted to a timed automaton or a network of time automata according to our definitions in Section 3. The output of the program is an XML representation of the UPPAAL model, which is ready to be employed and verified. The experimental tool and Orc examples appeared in this paper can be found on the web [9].

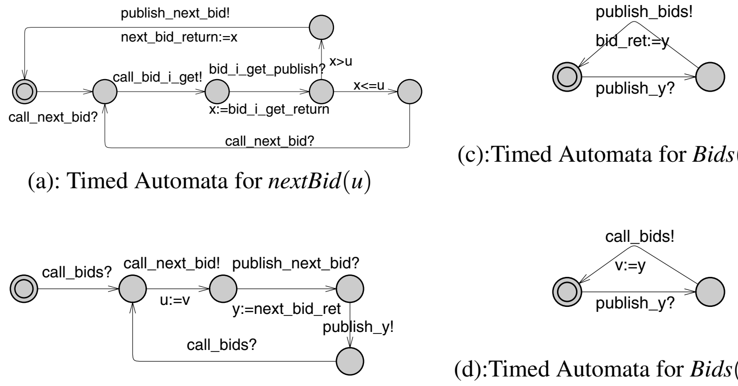
We briefly mention some of the implementation issues here. Because UPPAAL does not allow data to pass through channels, global variables are carefully defined to pass along the values, i.e., a *publish* event is always attached with an assignment to the respective global variable. An aggressive simplification procedure is applied whenever possible to simplify and optimize the constructed Timed Automata. For instance, when we apply Definition 6, if we are certain there is only one copy of *g* required, we may do the product of the two automata and remove the *publish* event given that it does not affect the rest of the model. We also try to minimize the number of clock variables by reusing the same ones as so to speed up the verification. However, the simplification and optimization remains as a challenging task and we may improve it by considering Orc laws.

Once the UPPAAL model is built, we may import it using UPPAAL and do verification. For example, it can be easily verified that the first Orc model of the dining philosophers can lead to deadlock. In our experiment, we created 5 philosophers and 5 fork instances. Afterwards we checked if the model is deadlock free using the following property: $A[]$ *not deadlock*. UPPAAL reports that the property does not hold for the system. A counterexample where all philosophers pick up their left fork can be found via random simulation. In the case that the first philosopher always picks up the right fork, we verify that the Orc model is deadlock-free and it satisfies properties like that no more than half of the philosophers can be eating at the same time etc.

4.2 Case Study: Orchestrating an Auction

In this subsection, we demonstrate the construction of the UPPAAL model from Orc as well as property checking through a typical web-based application, i.e., running an auction for an item. This example was originally presented in [17].

First, the item is advertised by calling site $Adv(v_0)$, which posts its description and a minimum bid price at a web site. Bidders put their bids on specific channels. In UPPAAL, a template called *Bidder* is built, which outputs a bid on channel *bid*. In general, there are multiple *Bidder*s. A *Multiplexor* is used to merge all the bids into a single channel, i.e., *bid*. The Timed Automata for the sites (like, $Adv(v_0)$, $PostNext(m)$ and $PostFinal(n)$) used in this example are not shown, because they have the same structure as the TA in Figure 8.



(a): Timed Automata for *nextBid*(u)

(c):Timed Automata for *Bids*(v) part 2

(b):Timed Automata for *Bids*(v) part 1

(d):Timed Automata for *Bids*(v) part 3

Fig. 9. Basic Sites in Auction Example

$$\begin{aligned}
&\mathit{Multiplexor}_i \mathrel{\widehat{=}} \mathit{bid}_i.\mathit{get} >y> \mathit{bid.put}(y) \gg \mathit{Multiplexor}_i\\
&\mathit{Multiplexor} \mathrel{\widehat{=}} \mathit{Multiplexor}_1 \mid \mathit{Multiplexor}_2 \mid \ldots \mid \mathit{Multiplexor}_i
\end{aligned}$$

Three variations on the auction strategy, $\mathit{Auction}_i(v)$ $(1 \leq i \leq 3)$ are considered. We start the auction by executing $z :\in \mathit{Auction}_i(V)$ where V is the minimum acceptable bid.

Non-terminating Auction . The first solution continually takes the next bid from channel *bid* which exceeds the current (highest) bid and posts it at a web site by calling *PostNext*.

$$\begin{aligned}
&\mathit{nextBid}(u) \mathrel{\widehat{=}} \mathit{bid.get} >x> \{(\mathit{if}(x > u) \gg \mathit{let}(x)) \mid (\mathit{if}(x \leq u) \gg \mathit{nextBid}(u))\}\\
&\mathit{Bids}(v) \mathrel{\widehat{=}} \mathit{nextBid}(v) >y> (\mathit{let}(y) \mid \mathit{Bids}(y))
\end{aligned}$$

Orc expression *nextBid*(v) returns the next bid from c exceeding v. The site call $\mathit{if}(x > v)$ returns a signal if $x > v$ and remains silent otherwise. *Bids*(v) returns a stream of bids from *bid* where the first bid exceeds v and successive bids are strictly increasing. The following strategy starts the auction by advertising the item, and posts successively higher bids at a web site. But the expression evaluation never terminates. The Timed Automata of *nextBid*(u) and *Bids*(v) are shown in Figure 9. The Timed Automata of *nextBid*(u) is simplified by combining the two if-condition automata with the main *nextBid*(u) TA, because the two conditions $(x > u)$ and $(x \leq u)$ are exclusive.

$$\mathit{Auction}_1(p) \mathrel{\widehat{=}} \mathit{Adv}(p) \gg \mathit{Bids}(p) >z> \mathit{PostNext}(z) \gg \mathbf{0}$$

Following the Timed Automata semantics defined in Section 3, $\mathit{Auction}_1(v)$ is interpreted as the automata in Figures 10 and 11.

By checking with UPPAAL, we can see that this version of the auction system is deadlock free, which means it never terminates. In this example, we assume that expression *let*(y) in *Bids*(v) is carried out fast enough so that there will not be an infinite number of threads of *let*(y). In addition to deadlock-freeness, we may verify properties like a bid is never lower than the minimum (see examples in Table 2).

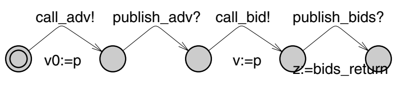


Fig. 10. TA for $Auction_1$ part 1

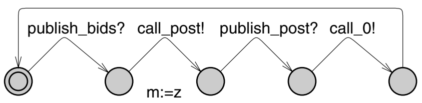


Fig. 11. TA for $Auction_1$ part 2

Table 2. Experiment Results

Orc	Property	Result	Time(s)	Remark
$Auction_1$	*A[] not deadlock*	true	20	Non-terminating.
$Auction_1$	*A[] not (PostNext.posted<250)*	true	3	No bid price lower 250.
$Auction_1$	*A[] not(old==0) imply new>old*	true	90	Price posted on the *PostNext* site keeps increasing.
$Auction_1$	*E<> PostNext.posted == 500*	true	1	Possible to post 500.
$Auction_2$	*A[] not deadlock*	false	1	Terminating.
$Auction_2$	*A[] PostFinal.postFinal imply Auc.c>=h*	true	150	Auction terminates after *h* time units.
$Auction_2$	*A[] PostFinal.final == 1000 imply Bidder10.bid == true*	true	10	The final bid comes from the respective bidder.
$Auction_3$	*A[] not deadlock*	false	1	Terminating.
$Auction_3$	*E[] not(PostNext.p1<h and PostNext.p1>0)*	true	60	It is not possible to post a highest bid before *h* time units.

In order to save space, the automata in the next two examples have been simplified whenever possible. Committed states are used to prevent undesired interleaving behaviors. For example, it is used to publish multiple signals at once for expressions like $let(x,y,z)$.

Terminating Auction. The previous program is modified so that the auction terminates if no higher bid arrives for *h* time units (say, *h* is an hour). The winning bid is then posted by calling *PostFinal*, and the goal variable is assigned the value of the winning bid. The expression $Tbids(v)$, where v is a bid, returns a stream of pairs $(x, flag)$, where x is a bid value, $x \geq v$, and *flag* is boolean. If *flag* is *true*, then x exceeds its previous bid, and if *false* then x equals its previous bid, i.e., no higher bid has been received in an hour.

$$\begin{aligned}
&Tbids(v) \mathrel{\widehat{=}} let(x, flag) \mid if(flag) \gg Tbids(x) \\
&\qquad \textbf{where } (x, flag) :\in nextBid(v) >y> let(y, true) \mid Rtimer(h) \gg let(v, false) \\
&Auction_2(v) \mathrel{\widehat{=}} Adv(v) \gg Tbids(v) >(x, flag)> \\
&\qquad \{if(flag) \gg PostNext(x) \gg \mathbf{0} \mid if(\overline{flag}) \gg PostFinal(x) \gg let(x)\}
\end{aligned}$$

In this auction, a new site call named *PostFinal* is added which is quite similar to *PostNext*. The difference between a non-terminating auction and a terminating auction is that a *time-out* (*h* time unit) process is added. As *time-out* (or *timed-interrupt*) is a typical timing behavior, we do define some templates to treat them specially and effectively. A list of typical composable timing patterns formally defined in terms of Timed Automata is available elsewhere in [8]. For example in Figure 12, we can use the

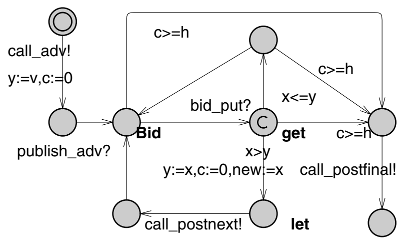


Fig. 12. Auction$_2$: Terminating Auction

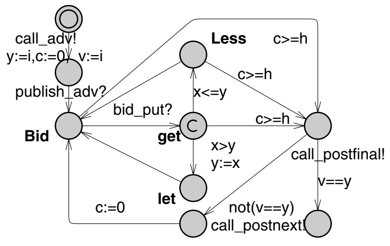


Fig. 13. Auction$_3$: Batch Processing

typical way of dealing with *time-out* in Timed Automata by adding a clock to record the time, as well as some clock constraints to guard the transitions. The constructed Timed Automata for *Auction*$_2$ is shown in Figure 12, in which c denotes the clock and h is a constant.

Batch Processing. The previous solution posts every higher bid as it appears in channel *bid*. It is reasonable to post higher bids only once each hour. Thus, the last solution collects the best bid over an hour and posts it. If this bid does not exceed the previous posting, i.e., no better bid has arrived in an hour, the auction is closed, the winning bid is posted and its value is returned as the result. In the interest of space, we skip the Orc model and the construction. The detail of the auction is available elsewhere in [17]. The constructed Timed Automaton is presented in Figure 13.

In the verification experiment of auction example using UPPAAL, we created 10 Bidders whose bid prices are from 200 to 1100, while the minimum bid price is 250. UPPAAL version 3.4 is installed on a machine running Windows XP with 3GHz Pentium 4 processor and 512MB memory. Some properties concerning all three auction strategies together the verification time are illustrated in Table 2.

4.3 Case Study: Purchase Order Handling

In this subsection, we present an example for handling purchase order, which was originally presented by Mistra and Cook [13].

$GetInv(custInfo, PO) \mathrel{\widehat{=}} ProduceInv(price, prodSchd) > inv > let(inv)$
  $\textbf{where}\ (price, prodSchd) :\in$
    $(let(x, y)$
      $\textbf{where}\ x :\in InitPriceCal(PO) \gg GetPrice(shpInfo) > x > let(x)$
        $y :\in (InitProdSchd(PO) \gg GetProdSchd(shpSchd) > y > let(y)$
          $\textbf{where}\ shpSchd :\in GetShpSchd(shpInfo)))$
    $\textbf{where}\ shpInfo :\in GetShpInfo(custInfo))$
$POHandling(custInfo, PO) \mathrel{\widehat{=}} MailInv(inv, custInfo)$
  $\textbf{where}\ inv :\in GetInv(custInfo, PO) \mid Rtimer(t) \gg let(error)$

On receiving the purchase order from a customer, the process initiates three tasks concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is mailed to the customer. If the invoice can not be generated within t time units, an error message is sent to the customer.

The purpose of this example is to show that the complex Orc expression can be represented by a clear UPPAAL model and the verification of time and data dependency. The UPPAAL model and property checking of this example are skipped (refer to [9]).

5 Conclusion and Future Works

The contribution of our work is threefold. Firstly, we defined an automata-based semantics for the Orc language, which allows a systematic construction of Timed Automata models from Orc models. Secondly, we explored ways of use UPPAAL to verify critical properties over Orc models. Lastly, we developed a tool to automate our approach.

There are some possible future works. One is better tool support of our approach, e.g., a graphical user interface for editing Orc models, hiding UPPAAL programs and visualizing counter examples if there are any, a better simplification and optimization strategy, etc. Another possible future work concerns the inadequate data passing capability of Orc, i.e., no complex data structure is supported. Therefore, we might provide a mechanism for introducing and manipulating data structures like arrays and tuples in our tool. The long term objective of this work is to investigate the relationship between process algebras and automata theories, e.g., provide theories and tools for applying automata-based model-checking to languages and notations based on process algebra.

Acknowledgements

The authors would like to thank Prof. Jayadev Misra for insightful discussion on the Orc language and pointing out relevant papers.

References

1. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. T. Amnell, A. David, and Y. Wang. A Real-Time Animator for Hybrid Systems. In J. W. Davidson and S. L. Min, editors, *LCTES*, volume 1985 of *Lecture Notes in Computer Science*, pages 134–145. Springer, 2000.
3. T. Amnell, E. Fersman, P. Pettersson, H. Sun, and Y. Wang. Code Synthesis for Timed Automata. *Nord. J. Comput.*, 9(4):269–300, 2002.
4. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Y. Wang. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*. Springer, 1995.
5. P. Brooke. *A Timed Semantics for a Hierarchical Desgn Notation.* PhD thesis, University of York, 1999.

6. W. R. Cook and J. Misra. A Structured Orchestration Language. 2005. Available for download at http://www.cs.utexas.edu/users/wcook/projects/orc.
7. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *In Hybrid System III: Verification and Control*, pages 208–219, 1996.
8. J. S. Dong, P. Hao, S. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In J. Davies, W. Schulte, and M. Barnett, editors, *ICFEM'04*, volume 3308 of *Lecture Notes in Computer Science*, pages 483–498. Springer, 2004.
9. J. S. Dong, Y. Liu, J. Sun, and X. Zhang. http://nt-appn.comp.nus.edu.sg/fm/orc, 2006.
10. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Automated Software Engineering 2003*, 2003.
11. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, 1992.
12. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
13. IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. BPEL4WS, Business Process Execution Language for Web Service version 1.1, 2003. http://www.siebel.com/bpel.
14. H. M. Lin and Y. Wang. A Proof System for Timed Automata. In J. Tiuryn, editor, *FoSSaCS*, volume 1784 of *Lecture Notes in Computer Science*, pages 208–222, 2000.
15. R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
16. R. Milner. *Communicating and Mobile Systems: the π Calculus*. Cambridge University Press, 1999.
17. J. Misra and W. Cook. *Computation Orchestration: A Basis for Wide-Area Computing*. To appear in the Journal of Software & Systems Modeling, 2006.
18. J. Misra, T. Hoare, and G. Menzel. A Tree Semantics of an Orchestration Language. In *M. Broy (ed.) Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, NATO ASI Series*, Marktoberdorf, Germany, August 2004.
19. G. G. Pu, X. P. Zhao, S. L. Wang, and Z. Y. Qiu. Towards the semantics and verification of BPEL4WS. In *International Workshop on Web Languages and Formal Methods*, UK, 2005.
20. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
21. S. Schneider and J. Davies. A Brief History of Timed CSP. *Theoretical Computer Science*, 138, 1995.
22. M. P. Singh and M. N. Huhns. *Service-Oriented Computing*. John Wiley & Sons, Ltd, 2005.
23. M. Sorea. TEMPO: A Model-checker for Event-recording Automata. In *Proceedings of Workshop on Real-time Tools*, August 2001.
24. A. Tiu. Model Checking for Pi-calculus Using Proof Search. In *CONCUR'05*, San Francisco, August 2005.

Appendix. Correctness Proof

This section presents the proof of the weak bi-simulation relation between the timed automata and the operational semantics of Orc. In this proof, the Orc expressions refer to a subset of Orc langauge that is regular, type-safe and with a finite number of threads (see Section 4 for details).

Definition 10. *Given an Orc expression, the transition system associated with the expression is* $(\mathcal{O}, o_0, \Sigma, \longrightarrow_1)$ *where* $\mathcal{O}$ *is the set of possible Orc configurations,* o_0 *is the initial given Orc expression,* Σ *is the alphabet which includes all events in the Orc semantics [17], and* $\longrightarrow_2$ *is the transition relation by the transition rules [17].*

Definition 11. *Given a Timed Automaton, the transition system associated with the automaton is* $(\mathcal{S}, s_0, \Sigma \cup \mathbb{T}, \longrightarrow_2)$ *where* $\mathcal{S} \mathrel{\widehat{=}} S \times V$ *is the set of all possible states. Each state is composed of a control state and a valuation of the clocks. The initial state* $s_0 = \langle i, v_0 \rangle$ *comprises the initial state i and a zero valuation* v_0. $\longrightarrow_2 \subseteq \mathcal{S} \times (\Sigma^\tau \cup \mathbb{T}) \times \mathcal{S}$ *comprises all possible transitions of the following two kinds:*

- *a time passing move* $\langle s, v \rangle \xrightarrow{\delta}_2 \langle s, v + \delta \rangle$.
- *an action execution* $\langle s, v \rangle \xrightarrow{a}_2 \langle s', v' \rangle$ *iff* $s \xrightarrow{a;\, X;\, \varphi} s'$. *That is, the clock interpretation meets the guard* ($v \models \varphi$), *and the new clock valuation satisfies:* $v'(x) = 0$ *for all* $x \in X$ *and* $v'(x) = v(x)$, *for all* $x \notin X$.

Definition 12. *For any* $o \in \mathcal{O}$ *and* $s \in \mathcal{S}$, $c \approx s$ *if and only if,*

- $\forall \alpha \in \Sigma, o \xrightarrow{\alpha}_1 o'$ *implies there exists* s' *such that* $s \xrightarrow{\alpha}_2 s'$, *and* $o' \approx s'$.
- $\forall \alpha \in \Sigma, s \xrightarrow{\alpha}_2 s'$ *implies there exists* o' *such that* $o \xrightarrow{\alpha}_1 o'$, *and* $o' \approx s'$.

Theorem 2. *Given an Orc expression Orc, let* $LTS_{Orc} \mathrel{\widehat{=}} (\mathcal{O}, o_0, \Sigma, \longrightarrow_2)$ *be the transition system associated with the expression. Let* $\mathcal{A}_{Orc}$ *be the corresponding Timed Automaton defined using definition 3 to 9 in the paper. Let* $LTS_{\mathcal{A}_{Orc}} \mathrel{\widehat{=}} (\mathcal{S}, s_0, \Sigma \cup \mathbb{T}, \longrightarrow_1)$ *be the transition system associated with the Timed Automaton.* $o_0 \approx s_0$.

Proof: The theorem can be proved by a structural induction on the Orc expressions. To abuse notations, we write $Orc \approx \mathcal{A}_{Orc}$ to mean $LTS_{Orc}.o_0 \approx LTS_{\mathcal{A}_{Orc}}.s_0$.

- $\mathbf{0}$: In Orc semantics, there is no transition rule for $\mathbf{0}$, so $LTS_{\mathbf{0}}$ is a single state transition system without any transitions. The same is $LTS_{\mathcal{A}_{\mathbf{0}}}$. Thus, $\mathbf{0} \approx \mathcal{A}_{\mathbf{0}}$.
- $let(z)$: In Orc semantics, the only transition for $LTS_{let(z)}$ is $let(z) \xrightarrow{publish_z}_1 \mathbf{0}$. It is also the only transition in the responding Timed Automaton. Thus, $let(z) \approx \mathcal{A}_{let(z)}$.
- $Rtimer(t)$: In Orc semantics [17], there is no transition rules for this basic site. However, it plays an important role in our work. After being called, the only transition allowed is time passing,

  $$Rtimer(t) \xrightarrow{\delta_{t_1}}_1 Rtimer(t - t_1);\ Rtimer(t) \xrightarrow{\delta_t}_1 \mathbf{0}$$

  The calling site is blocked until the t time units has elapsed. By definition 4 and 11, the Timed Automaton bi-simulates the site $Rtimer(t)$.
- The proof for fundamental sites *if* and *Signal* are skipped. There are no formal semantics defined for them. We can treat them as the normal site calls.
- Site call $M(P)$: According to Orc's operational semantics [17], the transitions in $LTS_{M(P)}$ are $M(P) \xrightarrow{call_{M(P)}}_1 ?k$ and $?k \xrightarrow{get_{M(P)}}_1 let(v)$. According to our definition 5, the two transitions have one-to-one correspondence to the transitions in the Timed Automaton shown in figure 2. In particular, $s_2 \approx let(v)$ and, therefore, $s_1 \approx ?k$ and, lastly, $s_i \approx M(P)$. Thus, $M(P) \approx \mathcal{A}_{M(P)}$.
- Sequential composition $f >x> g$: According to the operational semantics of Orc, the two transitions available for the sequential composition are:

$$f >x> g \xrightarrow{a}_1 f' >x> g \;\; \mathit{if} \;\; f \xrightarrow{a}_1 f'$$
$$f >x> g \xrightarrow{publish_v}_1 (f' >x> g) \mid [v/x].g \;\; \mathit{if} \;\; f \xrightarrow{publish_v}_1 f'$$

Assume $f \approx \mathcal{A}_f$ and $g \approx \mathcal{A}_g$. For every a such that if $f \xrightarrow{a}_1 f'$, there is a transition in $LTS_{f>x>g}$. Because $\mathcal{A}_{f>x>g}$ is $\mathcal{A}_f \parallel \mathcal{A}'_g$ (by definition 6), there is a corresponding transition in $\mathcal{A}_{f>x>g}$ because a is local to automaton $\mathcal{A}_f$ and by definition 2 the local actions are free to occur. Moreover, $f' \approx \mathcal{A}_{f'}$ by assumption. If $f \xrightarrow{publish_v}_1 f'$, then $f >x> g \xrightarrow{publish_v}_1 (f' >x> g) \mid [v/x].g$. By definition 6, there is a corresponding transition in $\mathcal{A}'_g$. As long as the number of *publish* events are finite, there is always a corresponding transition in one of the A'_g.

In the other direction, for every transition a from the initial state of $\mathcal{A}_{f>x>g}$, if a is a *publish* event, it must be a synchronization between $\mathcal{A}_f$ and one of the $\mathcal{A}'_g$. By assumption, there must be a transition $f \xrightarrow{a}_1 f'$. Therefore, there is a corresponding transition in $f >x> g \xrightarrow{publish_v}_1 (f' >x> g) \mid [v/x].g$. If a is a local event, then it must belong to $\mathcal{A}_f$ because the only transition in $\mathcal{A}'_g$ at its initial state is a synchronized *publish* event. There must be a corresponding transition in LTS_f and $LTS_{f>x>g}$. By induction, we conclude $f >x> g \approx \mathcal{A}_{f>x>g}$.

– Symmetric composition $f \mid g$: According to the operational semantics of Orc, the two transitions available for the sequential composition are:

$$f \mid g \xrightarrow{a}_1 f' \mid g \;\; \mathit{if} \;\; f \xrightarrow{a}_1 f'$$
$$f \mid g \xrightarrow{a}_1 f \mid g' \;\; \mathit{iff} \;\; g \xrightarrow{a}_1 g'$$

Therefore, f and g are interleaving. By definition 7, the corresponding Timed Automaton is defined as $\mathcal{A}_{f|g} \mathrel{\widehat{=}} \mathcal{A}_f \parallel \mathcal{A}_g$. The events in both f and g are renamed so that there is no synchronization between f and g. Assume $f \approx \mathcal{A}_f$ and $g \approx \mathcal{A}_g$. By definition 2 and the above, transitions rules, we conclude $f \mid g \approx \mathcal{A}_{f|g}$.

– Asymmetric composition f **where** $x :\in g$: According to the operational semantics of Orc, the two transitions available for the sequential composition are:

$$f \textbf{ where } x :\in g \xrightarrow{a}_1 f' \textbf{ where } x :\in g \;\; \mathit{if} \;\; f \xrightarrow{a}_1 f'$$
$$f \textbf{ where } x :\in g \xrightarrow{a}_1 [v/x].f \;\; \mathit{if} \;\; g \xrightarrow{a}_1 g'$$
$$f \textbf{ where } x :\in g \xrightarrow{a}_1 f \textbf{ where } x :\in g' \;\; \mathit{if} \;\; g \xrightarrow{a}_1 g' \;\mathit{and}\; a \neq !v$$

Form the transaction rules we can conclude the following three properties: 1) f and g run in parallel; 2) the first returned value of g is passed to f and g stops; 3) f is blocked if x is not available. From the three properties, the transition system $LTS_{f \textbf{ where } x:\in g}$ is the production of LTS_f and LTS_g, where they synchronized on the transition $publish_x$ and g is stopped after the synchronization. $LTS_{\mathcal{A}_{f \textbf{ where } x:\in g}}$ is exactly the same transition according to the definition 7, which uses the shared flag to stop the execution of g.

– Expression call $E(P) \mathrel{\widehat{=}} f$: According to the operational semantics of Orc, the transition available for expression call composition is: $E(P) \xrightarrow{\tau}_1 [P/x].f \;\mathit{iff}\; [\![E(x) \mathrel{\widehat{=}} f]\!] \in D$. The internal event τ acts as the initial event of the expression. It passes the input value to formal parameters. The equivalent event in the TA model is $call_E(P)$

event in the definition 9. The one-to-one mapping is shown in the following two transition systems.

$$\begin{aligned}
LTS_{E(P)} &\mathrel{\widehat{=}} (\{LTS_f.\mathcal{S} \cup o_0\}, \{LTS_f.\Sigma \cup \tau\}, o_0, \\
&\qquad \{LTS_f.\longrightarrow_1 \cup (o_0, \tau, LTS_f.o_0)\}) \\
LTS_{\mathcal{A}_{E(P)}} &\mathrel{\widehat{=}} (\{LTS_{\mathcal{A}_f}.\mathcal{S} \cup (i, v_0)\}, \{LTS_{\mathcal{A}_f}.\Sigma \cup \tau\}, (i, v_0), \\
&\qquad \{LTS_f.\longrightarrow_1 \cup ((i, v_0), \tau, (LTS_f.s_0.i, v_0)))\})
\end{aligned}$$

Therefore, we conclude that our Timed Automata semantics is sound.