Verification of Computation Orchestration via Timed Automata

Jin Song Dong, Yang Liu*, Jun Sun, Xian Zhang
School of Computing, National University of Singapore
{dongjs,liuyang,sunj,zhangxi5}@comp.nus.edu.sg

ABSTRACT

Recently, a promising programming model called *Orc* is proposed to support a structured way of orchestrating distributed web services. *Orc* is intuitive because it offers concise constructors to manage concurrency communication, time-outs, priorities, failure of sites or communication and etc. The semantics of *Orc* is also precisely defined.

However, there is no verification tool available for *Orc* to verify critical properties of *Orc* expressions. Instead of building one from scratch, we believe existing mature model-checker 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 can be systematically constructed from *Orc* models. The practical implication is that tool supports for TA, e.g. Uppaal, can be used to model check *Orc* models. An experimental tool is implemented to automate our approach.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic Execution*

General Terms

Language, Verification

Keywords

Orc, Timed Automata, Verification, Uppaal

1. INTRODUCTION

The prevalence of the internet and web services raises the request of service-oriented computing [20], which can invoke remote services, process the results and communicate results with other terminals. However, it is very difficult and complex to design a orchestrating system with currency and synchronization using the 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 [18, 9] is proposed for orchestrating distributed services in a structured manner. It abstracts all computations, web services and time control mechanism as site calls (See Section 2.1), which are implemented by primitive remote procedures. With this abstraction, it provides concise syntax for concurrent site call executions, threads synchronization and message passing. In addition, slow response and service failure can easily handled using timing site calls in Orc. Using Orc, complicated orchestrating problems can be easily understood and constructed without worrying about the programming details.

Orc is precise and elegant. Both operational semantics [18] and denotational semantics (a tree semantics [19]) are defined. However, as a new emerging language, there is no formal model-checking technique to systematically verify critical properties over the system modelled in Orc. Orc roots from traditional process algebras [16, 13, 17], so we believe that existing mature model-checkers can be reused instead of building one from scratch. In this work, 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. First of all, we define an executable model in Timed Automata [3] (TA) for Orc expressions, which conforms the semantics of the Orc language as defined in [9]. As a natural consequence, existing tool supports for Timed Automata, e.g. Uppaal [6], can be used for verification of Orc models. The dining philosopher example is used as a running example to demonstrate our approach. Moreover, we implement a tool to construct Uppaal models automatically from Orc model, as a demonstration that our approach can be fully automated.

Orc has a strong theoretical foundation in process algebras, particularly CCS [16], CSP [13] and π-calculus [17]. These process algebras provide fundamental models of concurrency in which processes communicate over channels. However, Orc is different from above process algebras that Orc per-

*Author for correspondence, fax: +65 6779 4580, phone: +65 6874 2834

mits 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 support, like FDR2 [22] for CSP, CWB-NC [8] for CCS and FO$\lambda^{\triangle\triangledown}$ [26] for π calculus. Because the absence of timing handling in the process algebras, none of these tools can model the timing aspects of complex systems. There are some process algebras with time extensions, for example, Timed CCS [28] and TCSP [23]. Unfortunately, there is no good model checker available[1].

Timed Automata [3] is a theory 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 TA, like Uppaal [6], KRONOS [10], TEMPO [24] and RED [27]. This gives the inspiration of this work. Clearly. the translation from Orc to TA will link the good modelling technique with sound verification tool support without repeating the work.

Business process orchestration languages, like XML based BPEL4WS [2], share many common elements with Orc. Both BPEL4WS and Orc treat a process and the services it orchestrates differently. Both languages provide mechanism for parallel execution and sequencing. Several recent work addresses the verification problem of BPEL4WS. Model-based verification of BPEL4WS [11, 15] models web services work flows using the Finite State Processes notation. Its supporting verification tool LTSA-WS [12] is also available. Another approach [21] tries to translate the self-defined semantics of BPEL4WS to TA, and the verification is done by Uppaal.

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 model in Timed Automata for each and every constructor in Orc. Section 4 demonstrates how Uppaal is used to verify the Orc languages via a case study. 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 at [9].

[1]To our knowledge, the only tool support for TCSP is the preliminary PVS encoding of TCSP in Brooke's PhD thesis [7]

$$
\begin{array}{lll}
D \in \mathit{Decl} & ::= & E(Q) \mathrel{\widehat{=}} f \\
f, g \in \mathit{Expression} & ::= & \mathbf{0} \\
& \| & M(P) \\
& \| & E(P) \\
& \| & f >x> g \\
& \| & f \mid g \\
& \| & f \textbf{ where } x :\in g
\end{array}
$$

$$
\begin{array}{lll}
p \in \mathit{Actual\ Parameter} & ::= & x \quad \text{– [Variable]} \\
& \| & c \quad \text{– [Constant]} \\
& \| & M \quad \text{– [Site]} \\
q \in \mathit{Formal\ Parameter} & ::= & x \quad \text{– [Variable]} \\
& \| & M \quad \text{– [Site]}
\end{array}
$$

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 is 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, like a web service implemented on the client's machine or 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 (d) where *ESPN* is a sports news service and d is a date, may download the news page for the specified date. Calling $\mathit{Email}(a, m)$ sends message m to address a, causing permanent change in the recipients 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 sequencing of site calls. For example, $\mathit{ESPN} > m > \mathit{Email}(a, m)$ will first call site *ESPN*, and name the returned value as m. After that $\mathit{Email}(a, m)$ is then called. If either site fails to respond, then the evaluation returns no value. The simpler notation $\gg$ without a parameter 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

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

Symmetric Parallel Operator Symmetric parallel operator | gives the power of multiple threads computation. Evaluation of $f \mid g$, creates two threads to compute f and g. The result from $f \mid g$ is the merge of these two streams in time order. If both threads produce values simultaneously, they are merged arbitrarily. Operator | is commutative and associative.

A particularly interesting expression is like $(\mathit{ESPN} \mid \mathit{BBC}) > m > \mathit{Email}(a, m)$. Here, the first part $(\mathit{ESPN} \mid \mathit{BBC})$ may publish multiple values, and for each value v, we call

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.
$Signal$	returns a signal immediately. It is same as $Rtimer(0)$.
$if(b)$	where b is boolean, returns a signal if b is true, and remains silent (no response) if false.

Table 1: Fundamental Sites

$Email(a, m)$ where m is set to v. Therefore, the evaluation can cause up to two emails to be sent, one with the value from $ESPN$ and the other from BBC.

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

Expression Definition An expression is defined by its name, a list of parameters which serve as its global variables, and an expression which serves as its body. As an example, consider the following restaurant reservation process, where the user is notified for the first acknowledgement received from the two restaurants, if any.

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

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

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

Orc does allow irregular expressions and divergence-like behaviors. Considering the following examples:

$$M \mathrel{\widehat{=}} let(x) \textbf{ where } x :\in let(0) \mid signal$$
$$N \mathrel{\widehat{=}} let(x) \textbf{ where } x :\in N$$

where what published by M is either of type integer (0) or simply a signal, and N never make a site call or make a signal and yet never terminates. In this work, we regard both kinds of Orc expressions as problematic ones.

Example: Dining Philosophers An example of using Orc is the classical dining philosophers problems.

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 $Fork_{i'}$. Philosopher i can eat only if it holds both left and right forks. A philosophers life cycle consists of following activities: acquire the two adjacent forks, eat, and release the forks. Because of the seating arrangement, neighboring philosophers can not eat simultaneously.

Each $Fork_i$ 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. 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.

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

The problem can be represented as:

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

It can be easily seen that this definition for dining philosophers can lead to deadlock. To avoid deadlock, philosophers should pick up their forks in a specific order: 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.

$$P_0 \mathrel{\widehat{=}} \\ Fork_1.get \gg Fork_0.get \gg Eat \gg \\ Fork_1.put \gg Fork_0.put \gg P_0$$

$$P_i \ (1 \leq i < N) \mathrel{\widehat{=}} \\ Fork_i.get \gg Fork_{i'}.get \gg Eat \gg \\ Fork_i.put \gg Fork_{i'}.put \gg P_i$$

2.2 Timed Automata and Uppaal

Timed Automata are finite state machines equipped with clocks. It is a formal notation to model the behavior 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 $\Phi(C)$ are defined by the following grammar:

$$\phi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \phi_1 \wedge \phi_2$$
where x is a clock variable and
c is a real number.

DEFINITION 2.1 (TIMED AUTOMATA). *A timed automaton $\mathcal{A}$ is a 6-tuple $\langle S, s_i, \Sigma, C, I, T \rangle$, where S is a finite set of states, s_i is the initial state, Σ is the alphabet, C is a finite set of clocks, $I : S \to \Phi(C)$ 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 a timed automaton, each node is associated with an invariant, while a transition is labelled with a guard (a constraint on clocks), a synchronization action 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 others keep their value. For example, Figure 1 illustrates some simple timed automata. Notice that a doubled-circle indicates an initial state.

Typically, for complex systems, a modelling would consist of a network of timed automata[2].

DEFINITION 2.2 (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* $\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$*, Each* $\mathcal{A}_i$ *is a timed automaton over its clocks. 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* $(s_1', e?, c', i', s_2') \in \mathcal{A}_j$. □

In this work, Uppaal [6] 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 a network of 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 [25]. 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

In this section, we define a Timed Automata semantics for Orc model, which allows us to systematically construct 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 purpose, for instance, synthesis of implementation [5], simulation [4], theorem proving [14] or more importantly formal verification [6].

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

In the following, the Timed Automata semantics for Orc expressions is formally defined. The dining philosopher example is used as a running example in this section.

DEFINITION 3.1 (ZERO SITE). *A zero site* **0** *is modelled as an automaton* $\mathcal{A}_\mathbf{0}$ *where* $S = \{s_i\}$ *contains only the initial state and* $\Sigma = \{call_\mathbf{0}\}$ *and* $C = \varnothing$ *and* $I = \varnothing$ *and* $T = \{(s_i, call_\mathbf{0}, \varnothing, true, s_i)\}$. □

A zero site is a site that can be called but never responds. Thus there is only one transition allowed, i.e. the site call of the zero site, as illustrated as the first automaton in Figure 1. Similarly, other fundamental sites are defined as timed automata as well, some of which are illustrated in Figure 1. The formal definition of the automaton for the fundamental site $Rtimer(t)$ is presented below as it plays the central role in the timing aspect of the orchestration.

DEFINITION 3.2 ($Rtimer(t)$). *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* $T = \{(s_i, call_{Rtimer(t)}, \{x\}, true, s_1), (s_1, get_{Rtimer(t)}, \varnothing, x = t, s_i)\}$. □

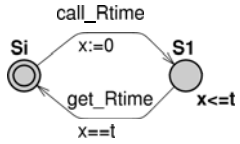


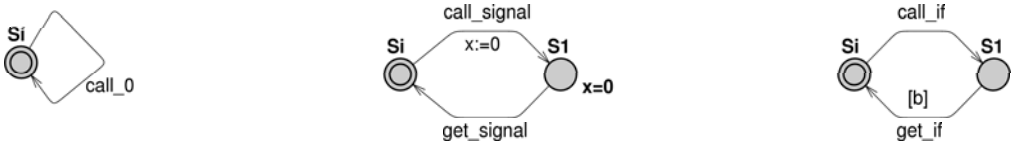
Figure 2: Fundamental Site: $Rtimer(t)$

The timed automaton is illustrated in Figure 2. 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 to program time-out or timed-interrupt. Therefore, 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.

DEFINITION 3.3 (SITE CALL). *A site call* $M(P)$ *is modelled as an automaton* $\mathcal{A}_{M(P)}$ *where* $S = \{s_i, s_1, s_2, s_3\}$ *and* $\Sigma = \{call_{M(P)}, get_{M(P)}, publish_{M(P)}\}$ *and* $C = \varnothing$ *and* $I = \varnothing$ *and*

$$T = \{ \begin{array}{l} (s_i, call_{M(P)}, \varnothing, true, s_1), \\ (s_1, get_{M(P)}, \varnothing, true, s_2), \\ (s_i, publish_{M(P)}, \varnothing, true, s_3)\} \end{array}$$

□



a: Zero Site b: Signal c: If

Figure 1: Fundamental Sites

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 3. This conforms the operational semantics of site call, i.e. the three steps of invocation, response, publication as defined in [9].

Figure 3: Timed Automaton for Site Call

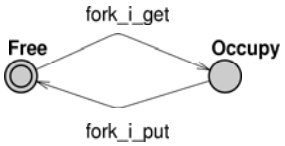


Figure 4: Timed Automaton for $Fork_i$

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 philosopher example are modelled as 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[3].

Definition 3.4 (Sequential Composition). *Let the automata network of g be $\mathcal{A}_g \widehat{=} \mathcal{A}_1 \parallel \ldots \mathcal{A}_n$. A sequential composition $f >x> g$ is modelled as a timed automata network $\mathcal{A}_{f>x>g} \widehat{=} \mathcal{A}_f \parallel \mathcal{A}'_g$ where, $\mathcal{A}'_g \widehat{=} (\mathcal{A}'_1 \parallel \ldots \mathcal{A}'_n)^k$ and for all $i : 1..n$, $\mathcal{A}'_i \widehat{=} \langle S, s_i, \Sigma, C, I, T \rangle$ 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 named $publish_x$ is defined to synchronize the publishing of a value of x and the receiving of the value[4]. 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 the Orc expressions, the first publishing signal, i.e. event *publish*, 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 forked and starts execution. In general, there would be infinite number of overlapping activation of the g component. However, if we assume the g part executes reasonably fast (and terminating), we may only need 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 philosopher example. Only one copy for each automaton is shown as that is all that needed in this case.

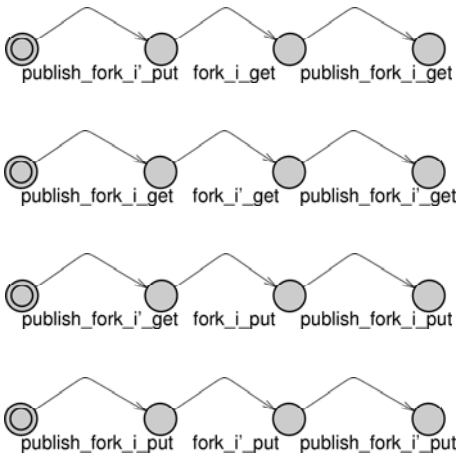


Figure 5: Network of Automata for $P_i (1 \leq i \leq N)$

[3]In Uppaal, it corresponds to a transition labelled with no channel event.

[4]In Uppaal, the actual value is passed through some shared variable since no data is attached in a channel communication.

DEFINITION 3.5 (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. In Orc, there are no communication between the f and g. For instance, 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 philosopher 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 [9], 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 3.6 (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 $\mathcal{A}_{f \textbf{ where } x:\in g} \mathrel{\widehat{=}} \mathcal{A}_f \parallel \mathcal{A}'_g$ where, $\mathcal{A}'_g = \mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_n$ and for all $i : 1..n$, $\mathcal{A}'_i \mathrel{\widehat{=}} \langle \mathcal{A}_i.S, \mathcal{A}_i.s_i, \mathcal{A}_i.\Sigma, \mathcal{A}_i.C, \mathcal{A}_i.I, T \rangle$ where*

$$T = \{(s_1, e, cl, g \wedge \mathit{flag}, s_2) \mid (s_1, e, cl, g, s_2) \in \{(s_1, \mathit{publish}_x, \mathit{flag} := \mathit{false}, cl, g \wedge \mathit{flag}, s_2) \mid (s_1, \mathit{publish}_x, cl, g \wedge \mathit{flag}, s_2) \in \mathcal{A}_i.T\}\}$$

□

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 implemented in a way so that it is local the automaton 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 $\mathit{publish}_x$ 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 is generic, there are many simplification and optimization to be performed on the constructed timed automata. For example, the P_i expression is modelled as automata in Figure 6.

DEFINITION 3.7 (EXPRESSION CALL). *An expression call is $E(P)$ with $E(P) \mathrel{\widehat{=}} f$ is modelled as the network of timed automata for f, i.e. $\mathcal{A}_f$.* □

For each parameter x of the expression call, a channel $\mathit{publish}_x$ 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 parameter gets its value (via synchronization on the respect channels). The publishing of the parameters may occur in any order.

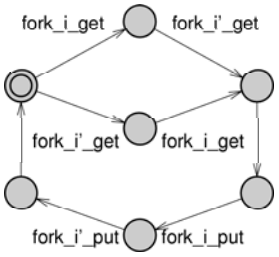


Figure 6: Timed Automata for P_i

For simple recursion where there is only one automaton instead of a automata network when we reach the recursion (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. We disallow these kinds of expressions for the sake of model checking.

The soundness of the interpretation is proved by showing that there a bi-simulation relation between the timed automata and the operational semantics of Orc. Formally,

THEOREM 3.8. *For all Orc expression f, $\mathcal{A}_f \cong \mathcal{O}_f$, where $\mathcal{O}_f$ is the state transition system constructed from the operational semantics of Orc in [18].* □

The theorem can be proved by a straightforward structural induction over our definitions and the operational semantics of Orc defined in [9]. We skip the detail in this version of the paper.

4. VERIFICATION USING UPPAAL

This section is devoted to a discuss on how to use tool supports 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 infinite number of automata, e.g. Definition 3.4. One evidence of possible infinite number of automata is that Orc in general allows irregular language (in terms of automata theory). Our target is therefore the subset of Orc langauge that are 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 irregular language. It is a known fact that such languages can not be expressed using automata. Therefore, they are out of league of 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 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 Hoare CSP's term.

4.1 Automated Construction

We developed an experimental tool to automatically construct Uppaal models from Orc models using XML and Java technology. There is not yet a standard interchange format for Orc. Therefore, we start with defining the syntax of Orc using XML schema. The XML schema and XML representation of Orc examples appeared in this paper can be found on the web *http://nt-appn.comp.nus.edu.sg/fm/orc*. The XML schema are carefully defined to express any Orc models in a structured text document as well as to force some well-formedness using XML constraints. Together with the XML schema, a parser and a transformation module is built using Java and SAX parser [1] to parse XML representations of Orc and construct respective Uppaal model automatically. The output of the program is an XML representation of the Uppaal model, which is readily to be employed and verified in Uppaal. The Uppaal model construction flowchart is shown in Figure 7.

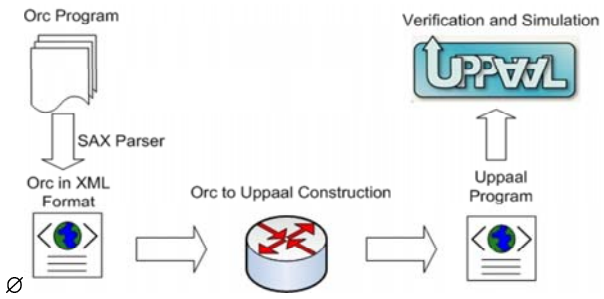


Figure 7: Automated Construction Flowchart

We briefly mention some of the implementation issues here. Because Uppaal does not allow data 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 somehow optimize the constructed Timed Automata. For instance, when we apply Definition 3.4, 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 as well try to minimize the number of clock variable via reusing the same as so to speed up the verification. However, the simplification and optimization remains as a challenge task and we may improve it by considering Orc laws.

Once the Uppaal 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 3 philosopher and 3 fork instances. Afterwards we checked for the deadlock free property using the following property: `A[] not deadlock`. Uppaal reports that the property does not hold for the system.

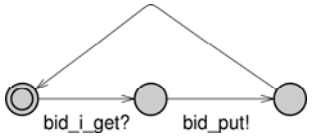


Figure 8: Timed Automata for $Multiplexor_i$

A counterexample where all philosophers pick up their left fork is presented. Whereas in the case that the first philosopher always picks up the right fork, we verify that the Orc model is both deadlock-free and that no more than half of the philosophers can be eating at the same time via checking the following: `A[] not (p1.eating and p2.eating ...)`

4.2 Case Study: Orchestrating an auction

In this subsection, we demonstrate the construction of 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 is originally presented in [18].

First, the item is advertised by calling site Adv, which posts its description and a minimum bid price at a web site. The corresponding Timed Automata for Adv is shown in Figure 9(a). Bidders put their bids on specific channels. In the Uppaal, a template called $Bidder$ is built, which outputs a bid on channel bid. We use the value assignment to realize the value passing through channels. Figure 9(b) illustrates the Uppaal model for $Bidder$. 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 $Multiplexor$ is illustrated in Figure 8.

$$Multiplexor_i \mathrel{\widehat{=}} bid_i.get >y> bid.put(y) \gg Multiplexor_i$$
$$Multiplexor \mathrel{\widehat{=}} (\mid i :: Multiplexor_i)$$

Three variations on the auction strategy, $Auction_i(v), i \in 1..3$ in considered. We start the auction by executing

$$z :\in Auction_i(V)$$

where $1 \leq i \leq 3$ and V is the minimum acceptable bid.

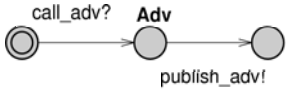
• Non-terminating auction

The first solution continually takes the next bid from channel c which exceeds the current (highest) bid and posts it at a web site by calling $PostNext$. Below, $nextBid(v)$ returns the next bid from c exceeding v. The site call $if(x > v)$ returns a signal if $x > v$ and remains silent otherwise.

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

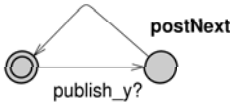
Below, $Bids(v)$ returns a stream of bids from bid where the first bid exceeds v and successive bids are strictly increasing.

$$Bids(v) \mathrel{\widehat{=}} nextBid(v) >y> (let(y) \mid Bids(y))$$



(a): *Adv* site call

(b): $Bidder_i$



(c): *PostNext* site call

Figure 9: Basic Sites in Auction Example

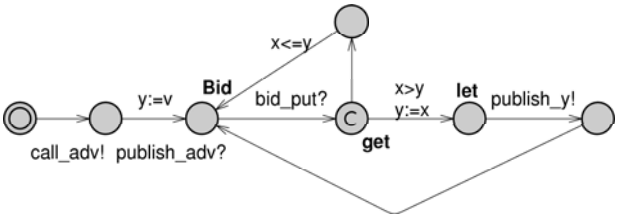


Figure 10: Timed Automata for $Auction_i$

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.

$$Auction_1(v) \mathrel{\widehat{=}} Adv(v) \gg Bids(v) >y> PostNext(y) \gg 0$$

PostNext is a site call which posts the current highest bid at a web site, which is shown in Figure 9(c).

Following the Timed Automata semantics defined in Section 3, Orc expression $Auction_1(v)$ is interpreted as automata in Figure 10. Notice that in order to save space, the automata have been simplified whenever possible. By checking with *Uppaal*, we can see that this version of the auction system is deadlock free, which means it is never terminating. In this example, we do assume that expression $let(y)$ is carried out fast enough so that there will not be 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.

• Terminating auction

We modify the previous program 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.

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.

$$\begin{array}{l} Tbids(v) \mathrel{\widehat{=}} \\ \quad let(x, flag) \mid if(flag) \gg Tbids(x) \\ \quad \textbf{where} \\ \quad\quad (x, flag) :\in \\ \quad\quad\quad nextBid(v) >y> let(y, true) \\ \quad\quad\quad \mid Rtimer(h) \gg let(v, false) \end{array}$$

The full auction is given by

$$\begin{array}{l} Auction2(v) \mathrel{\widehat{=}} \\ \quad Adv(v) \gg Tbids(v) \\ >(x, flag)> \\ \quad \{ if(flag) \gg PostNext(x) \gg \mathbf{0} \\ \quad\quad \mid if(flag) \gg PostFinal(x) \gg let(x) \\ \quad \} \end{array}$$

In this auction, a new site call named *PostFinal* (Figure 11) is added which is quite similar to *PostNext*.

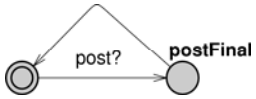


Figure 11: PostFinal site call

The difference between non-terminating auction and terminating auction is that a *timeout* (h time unit) process is added. We can use the normal way of dealing with *timeouts* in Timed Automata by adding a clock to record the time, as well as some clock constraints to guard the transitions. The translated Timed Automata for $Auction_2$ is shown in Figure 12, in which c denotes the clock and h is a constant.

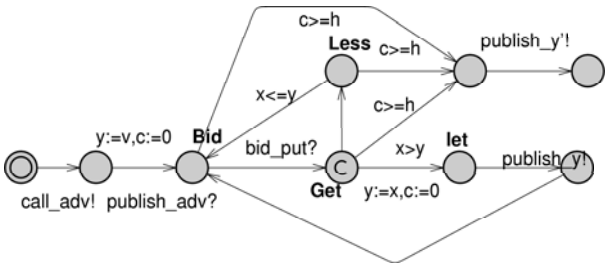


Figure 12: Auction$_2$: Terminating Auction

Orc Expression	Property	Result	Verification Time

In our example, we created 4 Bidders whose bid prices are 200, 300, 400 and 500, while the minimum bid price is 250. By using Uppaal, we checked the following properties:

- This auction will eventually terminate by going to state *PostNext*.
- If the fourth bidder with bid price 500 bids at anytime, he will eventually win this auction.
- Whenever the bidder with bid price 200 bids, it will never be posted in the PostNext site call.

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 a better tool support of our approach, e.g. a graphical user interface for editing Orc model, hiding Uppaal program and Visualizing counterexample if there is any, a better simplification and optimization strategy, etc. Another possible future work concerns about 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 in our tool. The long term objective of this work is to investigate the relationship between process algebras with automata theories, e.g. provide theories and tools for applying automata-based model-checking to languages and notations based on process algebra.

Acknowledgement

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

6. REFERENCES

[1] SAX http://www.saxproject.org/.

[2] BPEL4WS, Business Process Execution Language for Web Service, 2003.

[3] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[4] T. Amnell, A. David, and W. Yi. 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.

[5] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code Synthesis for Timed Automata. *Nord. J. Comput.*, 9(4):269–300, 2002.

[6] 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*, pages 232–243. Springer, 1995.

[7] P. Brooke. *A Timed Semantics for a Hierarchical Desgn Notation.* PhD thesis, University of York, 1999.

[8] R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Sci. Comput. Program.*, 42(1):39–47, 2002.

[9] W. R. Cook and J. Misra. A Structured Orchestration Language. In *Submitted for publication*, 2005.

[10] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *In Hybrid System III: Verification and Control*, pages 208–219, 1996.

[11] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *ASE*, 2003.

[12] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: A Tool for Model-based Verification and Validation of Web Service Compositions in Eclipse. In *EclipseCon 2005*, San Francisco, CA, 2005.

[13] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[14] H. Lin and W. Yi. A Proof System for Timed Automata. In J. Tiuryn, editor, *FoSSaCS*, volume 1784 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2000.

[15] J. Magee and J. Kramer. Compatibility Verification for Web Service Choreography. In *IEEE ICWS*, San Diego, CA, July 2004.

[16] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[17] R. Milner. *Communicating and Mobile Systems: the π Calculus.* Cambridge University Press, 1999.

[18] J. Misra and W. Cook. *Computation Orchestration: A Basis for Wide-Area Computing.* NATO ASI Series, 2005.

[19] J. Misra, T. Hoare, and G. Menzel. A Tree Semantics of an Orchestration Language. In *Lecture Notes for NATO summer school*, August 2004.

[20] M.P.Singh and M.N.Huhns. Service-Oriented Computing. 2005.

[21] G. Pu, X. Zhao, S. Wang, and Z. Qiu. Towards the semantics and verification of BPEL4WS. In *International Workshop on Web Languages and Formal Methods*, Newcastle, UK, 2005. WLFM 2005.

[22] A. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[23] S. Schneider and J. Davies. A Brief History of Timed CSP. *Theoretical Computer Science*, 138, 1995.

[24] M. Sorea. TEMPO: A model-checker for event-recording automata. In *Proceedings of Workshop on Real-time Tools*, August 2001.

[25] 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, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[26] A. Tiu. Model checking for pi-calculus using proof search. In *CONCUR*, San Francisco, August 2005.

[27] F. Wang. Symbolic verification of complex real-time systems with clock restriction diagram. In *Proceedings of international conference on Formal Techniques for Networked and Distributed Systems*, 2001.

[28] W. Yi. CCS + time = an interleaving model for real time systems. In *Proceedings of the 18th international colloquium on Automata, languages and programming*, pages 217 – 228, Madrid, Spain, 1991. Springer-Verlag New York, Inc.