1-2014

Towards verification of computation orchestration

Jin Song DONG

Yang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Xian ZHANG

Citation

DONG, Jin Song; LIU, Yang; SUN, Jun; and ZHANG, Xian. Towards verification of computation orchestration. (2014). *Formal Aspects of Computing*. 26, (4), 729-759.
Available at: https://ink.library.smu.edu.sg/sis_research/4977

DOI 10.1007/s00165-013-0280-9
BCS © 2013
Formal Aspects of Computing (2014) 26: 729–759

Formal Aspects of Computing

Towards verification of computation orchestration

Jin Song Dong[1], Yang Liu[2], Jun Sun[3] and Xian Zhang[1]

[1] School of Computing, National University of Singapore, Singapore, Singapore
[2] School of Computer Engineering, Nanyang Technological University, Singapore, Singapore
[3] Singapore University of Technology and Design, Singapore, Singapore

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 Web Services or communication and so forth. The semantics of Orc is precisely defined. However, there is no *automatic* verification tool available to verify critical properties against Orc programs. Our goal is to verify the orchestration programs (written in Orc language) which invoke web services to achieve certain goals. To investigate this problem and build useful tools, we explore in two directions. Firstly, we define a Timed Automata semantics for the Orc language, which we prove is semantically equivalent to the operational semantics of Orc. Consequently, Timed Automata models are systematically constructed from Orc programs. The practical implication is that existing tool supports for Timed Automata, e.g., UPPAAL, can be used to simulate and model check Orc programs. An experimental tool has been implemented to automate this approach. Secondly, we start with encoding the operational semantics of Orc language in Constraint Logic Programming (CLP), which allows a systematic translation from Orc to CLP. Powerful constraint solvers like CLP($\mathcal{R}$) are then used to prove traditional safety properties and beyond, e.g., reachability, deadlock-freeness, lower or upper bound of a time interval, etc. Counterexamples are generated when properties are not satisfied. Furthermore, the stepwise execution traces can be automatically generated as the simulation steps. The two different approaches give an insight into the verification problem of Web Service orchestration. The Timed Automata approach has its merits in visualized simulation and efficient verification supported by the well developed tools. On the other hand, the CPL approach gives better expressiveness in both modeling and verification. The two approaches complement each other, which gives a complete solution for the simulation and verification of Computation Orchestration.

Keywords: Orc, Web service orchestration, Verification, Timed automata, UPPAAL, Constraint logic programming, CLP($\mathcal{R}$)

This work is a substantial extension of the publicaton [DLSZ06] in 8th International Conference on Formal Engineering Methods, 2006.
Correspondence and offprint requests to: Y. Liu, E-mail: yangliu@ntu.edu.sg.

1. Introduction

The prevalence of the Internet and Web Services raises the request of Service-Oriented Computing (SOC) [SH05], which can invoke remote services, process the results and communicate results with other terminals. However, it is very challenging to design an orchestration system with concurrency and synchronization using practical programming languages because these traditional languages use threads for concurrency and semaphores for synchronization. Even the high-level libraries using channel and working pool have to be built up based on these primary elements.

Recently, a promising programming language Orc [MC07, CM05] 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 understood and constructed without worrying about the programming details.

Orc is precise and elegant. Both operational semantics [MC07, WKCM08] and denotational semantics (a tree semantics [MHM04] and a trace semantics [WKCM08]) are defined. However, as an emerging language, there are only limited formal verification mechanisms to systematically verify critical properties over systems modeled in Orc. The process of formal verification can prove that a system does not have defects or does satisfy desirable properties, which is very important for software systems. Critical system requirements like safety properties and liveness properties play important roles in the system specification, development and testing. In the context of Web Services, for example, it is critical to verify that a composite service upholds critical properties (e.g., ensuring that a credit card is only debited once per transaction, or not executing the order to send the merchandize until the goods are paid for). In this article, we address the verification problem of the Orc language. Our aim is to verify the orchestration programs (written in Orc language) which invoke web services to achieve certain goals. The critical properties to be verified include safety, liveness or even timing properties, of Orc programs.

Our first approach starts with defining an executable model in Timed Automata [AD94] for Orc programs, which conforms to the operational semantics of Orc language defined in [WKCM08]. The practical implication is that existing tool support for Timed Automata can be used for verification of Orc programs. We target at the state-of-the-art real time system modeling tool, UPPAAL [LPW97], which provides an interactive user interface for the real-time system simulation and verification. Moreover, we implement a tool to construct UPPAAL models automatically from Orc programs.

This approach not only successfully demonstrates a solution for Web Service orchestration verification, but also shows the possibility to verify timed process algebras in general based on Timed Automata. However, further investigation reveals that Timed Automata and UPPAAL has several limitations. Firstly, constrained by the Timed Automata theories, this approach can only handle a subset of Orc language that is regular, type-safe and with a finite number of threads. Secondly, model checkers for Timed Automata may not be optimal for the translated models from Orc programs. Therefore, the simulation and verification may be slow for large and complex systems. Finally, UPPAAL supports only a restrictive subset of timed CTL (Computation Tree Logic) [LPY95]. Limited by the nature of Timed Automata, these problems are hard to tackle.

Our second approach is a method based on Constraint Logic Programming (CLP) [JM94]. CLP is designed for mechanized verification using constraint solving, which has been successfully applied to model programs and transition systems for the purpose of verification. Our approach starts with a systematic encoding of the operational semantics of Orc [WKCM08] into CLP, which gives an equivalent CLP models. Then we develop CLP rules for verifying different kinds of properties. As the result, we can use powerful constraint solver like CLP($\mathcal{R}$) [JMSY92] to prove safety properties (e.g., reachability and deadlock-freeness), lower or upper bound of a time interval, etc. Counterexamples are generated when properties are falsified. Furthermore, the stepwise simulation can be displayed automatically from the counterexample traces. With more expressiveness power (compared to Timed Automata approach), this approach can handle all the Orc language structures.

We use an auction site example [MC07] to demonstrate our approaches. The dining philosopher example is used as a running example in the article to illustrate the concepts. Our experimental results demonstrate the effectiveness of the two approaches. The comparison of the two approaches shows that CLP implementation is faster and can support more properties as we expected.

1.1. Related works

Orc has a strong theoretical foundation in process algebras, particularly in CCS [Mil89], CSP [Hoa85] and π-calculus [Mil99]. These process algebras provide fundamental models of concurrency in which processes communicate over channels. Orc is different from the above process algebras as Orc permits integration of arbitrary components (sites) in a computation. More importantly, Orc has quantitative timing control to handle service failures. Traditional process algebras have well established model checking theories and tool supports, e.g., FDR2 [Ros97] and PAT [SLDP09, LSD10, LSD11] for CSP# [SLDC09], and FO$\lambda^{\Delta\nabla}$ [Tiu05] for π-calculus. Due to lack of quantitative timing support, none of these tools can model and verify timing aspects of complex systems. There are process algebras with time extensions, e.g., Timed CSP [SD95] and Stateful Timed CSP [SLD$^+$13]. To the best of our knowledge, there are few verification supports for Timed CSP, e.g., the theorem proving approach documented in [Bro99, GG09], the translation to UPPAAL models [DHQ$^+$04, DHQ$^+$08] and the approach based on constraint solving [DHSZ06].

There are a number of works on the verification of Orc programs. In [DLSZ06], we presented the first approach to verify Orc programs by encoded (asynchronous) operational semantics of Orc [MC07, MHM04, KCM06] using Timed Automata. However, this semantics misses the quantitative timing aspects, which are introduced later in [WKCM08]. In this work, we update our semantics encoding in Timed Automata according to [WKCM08]. Other semantics translation approaches include works like [BMT06, AM07, AM08, AM10]. In [BMT06], Bruni, Melgratti and Tuosto encode Orc in Petri nets and the join calculus. However, there is no verification support for this work. In [AM07, AM08, AM10], a rewriting semantics of Orc is defined, which is proved to be semantically equivalent to the operational semantics of Orc. Models in rewrite theory are then constructed from Orc programs manually, and Maude, which is a simulator and model checker for rewrite theory, is used to simulate and model check Orc programs. This approach is manual and their semantics based on clock-tick events (i.e., time must be modeled discretely). Compared with these works, our approaches (including the second approach described below) are automatic and support the dense-time semantic model. Furthermore, our second approach allows a systematic encoding the semantics of Orc language in CLP. CLP has been successfully applied to model programs and transition systems for the purpose of verification [GP97, JSV04]. Therefore, various properties (e.g., safety properties, lower or upper bound of a time interval, etc.) can be verified.

As an orchestration language, Orc is related to WS-BPEL (Web Service Business Process Execution Language) [OAS07]. WS-BPEL models business processes by specifying the work flows of carrying out business transactions, which shares many common elements with Orc. Both WS-BPEL and Orc orchestrate Web Services by using process composition (sequential and parallel) and communication (synchronous and asynchronous). However they are different in several ways. WS-BPEL has a rich set of the language structures to ease the process design. Orc's concise syntax allows the reuse of the process definitions. WS-BPEL has variables to store the state of the communications and is able to receive calls from client web services. Orc is abstract as it focuses on process and communication. Most importantly, Orc has a well-defined semantics.

Our work is closely related to the works on verification of Web Service orchestrations, which are quite recent, but have attracted considerable attentions. Some related works on the WS-BPEL verification are listed as follows. Model-based verification of WS-BPEL [FUMK03, FUMK06, FEK$^+$07, Fos08a] models Web Services work flows using the notion of Finite State Processes, which is then verified using their tool LTSA-WS [FUMK06] (and later WS-Engineer [Fos08b]). Nakajima [Nak05] proposed a method to extract the behavioral specification from a WS-BPEL process and to analyze it by using the SPIN model checker. A finite state automaton extended with variable annotations (definitions and updates) is used as an intermediate representation. In [PZWQ06], Pu and his colleagues defined an operational semantics for WS-BPEL as well as a transformation from WS-BPEL to Timed Automata. Comparing with [PZWQ06], our first approach focuses on Orc language, which has different characteristics from WS-BPEL. Firstly, Orc uses sites to express all the computations, where timers and conditional choice are treated as basic site calls. Secondly, the special compositional pattern *asymmetric parallel composition* is unique in Orc language. This operator is a basic syntax block in Orc to express different concepts like time-out, priority, nondeterministic choice, iterative process, parallel-or and so on [WKCM08]. Lastly, Orc has rigorous semantics, therefore we can prove the translation to TA is sound. Translating Orc to TA is not just an arithmetic task by repeating [PZWQ06]. We studied how to express different syntax and semantics using TA in a correct way. Other works use different computational models for verifying WS-BPEL processes. In [SMS05], a Petri Net semantics is provided for WS-BPEL. The net resulting from the translation is then validated with the LoLA model checking tool [Sch00]. The SENSORIA project [WHA$^+$08, WCG$^+$06, WDG$^+$07] develops various approaches on modeling and verification of SOC, where foundational theories, techniques and methods are fully integrated in a pragmatic software engineering approach. In [FGV04, FV06], an execution semantics for (an early version of) BPEL has been provided in terms of Abstract State Machines (ASMs). A more general framework for modeling Web Services based on ASM are proposed in [BT08b, BT08a]. Since ASMs provide a rigorous meaning to abstract code, for the verification and validation of properties of ASMs one can adopt every appropriate accurate method, without being restricted to mechanical (theorem proving or model checking) techniques. Ait-Sadoune and Ait-Ameur [ASAA08, ASAA09] proposed a proof and refinement based approach for the formal representation, verification and validation of web services compositions using the Event B method. They also developed the BPEL2B translator that automates the translation of BPEL into Event B models.

The rest of the article is organized as follows. Section 2 introduces the syntax and semantics of the Orc language and gives sample models. Sections 3 and 4 present our two approaches. Each of these two sections includes: background knowledge of the Timed Automata and CLP respectively, Orc program definitions and encodings in the target framework, verification solutions of the Orc language. Section 5 uses some examples to show the difference between the two approaches and compare the performance based on the experiment results. Section 6 compares the two different approaches in details. Finally, Sect. 7 concludes the article with possible future work.

2. Orchestration language Orc

This section presents the syntax and semantics of Orc. Formal definitions of Orc semantics can be referred in [CM05].

2.1. Syntax

Let E be an expression name; M be a site name; x be a variable; m be a value. The syntax of the Orc language is defined as follows[1]

$$\begin{array}{lll} D \in \mathit{Declaration} & ::= & E(\bar{x}) \mathrel{\widehat{=}} f \\ f, g \in \mathit{Expression} & ::= & M(\bar{p}) \parallel E(\bar{p}) \parallel f \;>x>\; g \parallel f \mid g \parallel f \;<x<\; g \\ p \in \mathit{Actual} & ::= & x \parallel m \end{array}$$

Declaration $E(\bar{x}) \mathrel{\widehat{=}} f$ defines expression E whose formal parameter list is $\bar{x}$ and body is expression f. We assume that only variables $\bar{x}$ are free in f. An expression is either an elementary expression or a composition of two expressions. An elementary expression is either a site call $M(\bar{p})$, or an expression call $E(\bar{p})$. An actual parameter p may be a variable x or a value m, and $\bar{p}$ is a list of actual parameters. If the parameter list is empty in $M(\bar{p})$ or $E(\bar{p})$, we simply write M and E. Orc has three composition operators: (1) $f \;>x>\; g$ for sequential composition, (2) | for symmetric parallel composition, and (3) $f \;<x<\; g$ for asymmetric parallel composition.

2.1.1. *Site*

The basic element of Orc programs 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 site 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.

2.1.2. *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

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

[1] Previous presentations of Orc have used the notation f **where** $x :\in g$ instead of $f \;<x<\; g$.

Table 1. Fundamental sites

0	Never responds. It can be used to terminate a computation.
$let(x, y, \ldots)$	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 a Boolean expression, returns a signal if b is true, and remains Silent (no response) if false.
$Signal$	Returns a signal immediately. It is the same as $Rtimer(0)$.

2.1.3. *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 timed 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 $Email(a, v)$ 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*.

2.1.4. *Asymmetric parallel operator*

The asymmetric parallel operator $<x<$ is used to prune portions of a computation selectively: $Email(a, m) < m < (Google(w) \mid Yahoo(w))$ 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 the evaluation of this expression is then terminated. After that, $Email(a, m)$ is unblocked and executed.

2.1.5. *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.

$$Reservation(t) \mathrel{\widehat{=}} Notify(x) <x< (R1(t) \mid R2(t))$$

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 philosopher example We use the classical dining philosopher problem [MC07] to demonstrate a complete Orc program. 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 the fork to his right is $Fork_{i'}$ where $i' = (i + 1)\ mod\ N$. Philosopher i can eat only if he 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 cannot eat simultaneously.

Each $Fork_i$ is modeled 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 denote sending a signal along the channel and $Fork_i.get$ to denote getting 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 as the completion of eating.

$$P_i \mathrel{\widehat{=}} ((let(x, y) \gg Eat_i \gg Fork_i.put \gg Fork_{i'}.put \gg P_i) <y< Fork_{i'}.get) <x< Fork_i.get$$

$$\frac{[\![E(\bar{q}) \mathrel{\widehat{=}} f]\!] \in D}{E(\bar{p}) \overset{0,\tau}{\rightarrow} [\bar{p}/\bar{q}].f}\ [\,\textsc{Def}\,] \qquad \frac{f \overset{t,a}{\rightarrow} f' \quad a \neq !m}{f > x > g \overset{t,a}{\rightarrow} f' > x > g}\ [\,\textsc{Seq1N}\,]$$

$$\frac{u \in \Sigma(M,m)}{M(m) \overset{0,\tau}{\rightarrow} ?u}\ [\,\textsc{SiteCall}\,] \qquad \frac{f \overset{t,!m}{\rightarrow} f'}{f > x > g \overset{t,\tau}{\rightarrow} (f' > x > g) \mid [m/x].g}\ [\,\textsc{Seq1V}\,]$$

$$\frac{(t,m) \in u}{?u \overset{t,!m}{\hookrightarrow} \mathbf{0}}\ [\,\textsc{SiteRet}\,] \qquad \frac{f \overset{t,a}{\rightarrow} f'}{f < x < g \overset{t,a}{\rightarrow} f' < x < g^t}\ [\,\textsc{Asym1}\,]$$

$$\frac{f \overset{t,a}{\rightarrow} f'}{f \mid g \overset{t,a}{\rightarrow} f' \mid g^t}\ [\,\textsc{Sym1}\,] \qquad \frac{g \overset{t,!m}{\rightarrow} g'}{f < x < g \overset{t,\tau}{\rightarrow} [m/x].f^t}\ [\,\textsc{Asym2V}\,]$$

$$\frac{g \overset{t,a}{\rightarrow} g'}{f \mid g \overset{t,a}{\rightarrow} f^t \mid g'}\ [\,\textsc{Sym2}\,] \qquad \frac{g \overset{t,a}{\rightarrow} g' \quad a \neq !m}{f < x < g \overset{t,a}{\rightarrow} f^t < x < g'}\ [\,\textsc{Asym2N}\,]$$

Fig. 1. Operational semantics of Orc

The dining philosopher problem can be represented as:

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

This definition of dining philosopher 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 forks first and then their right forks, and P_0 picks up the right fork and then the left fork.

$$\begin{array}{l}
P'_0 \mathrel{\widehat{=}} Fork_1.get \gg Fork_0.get \gg Eat_0 \gg Fork_1.put \gg Fork_0.put \gg P'_0 \\
P'_i(1 \leq i < N) \mathrel{\widehat{=}} Fork_i.get \gg Fork_{i'}.get \gg Eat_i \gg Fork_i.put \gg Fork_{i'}.put \gg P'_i \\
DP' \mathrel{\widehat{=}} P'_0 \mid P'_1 \mid \ldots \mid P'_{N-1}
\end{array}$$

Timed vending machine In this example, a user may insert some coins and then make a choice between coffee or tea. Once the choice is made, the vending machine dispatches the corresponding drink. Alternatively, the user may ask the machine to releasex the coins and walk away. If the user idles more than 10 seconds after the coin is inserted, the machine will release the coins.

$$\begin{array}{ll}
Select & \mathrel{\widehat{=}} ((if(flag == 1) \gg reqrelease \gg release \gg Rtimer(2) \gg Signal) \\
& \quad \mid (if(flag == 2) \gg coffee \gg Rtimer(3) \gg dispatch\_coffee \gg Signal) \\
& \quad \mid (if(flag == 3) \gg tea \gg Rtimer(2) \gg dispatch\_tea \gg Signal)) < flag < (let(1) \mid let(2) \mid let(3)) \\
Timeout & \mathrel{\widehat{=}} let(z) < z < (Select \mid Rtimer(10) \gg release \gg Signal) \\
TVM & \mathrel{\widehat{=}} coin \gg Timeout \gg TVM
\end{array}$$

Note that two patterns [WKCM08] are used here. Nondeterministic choice in *Select* expression is expressed using asymmetric composition such that the choice of a first value is nondeterministic if several values are published simultaneously. Time-out in *Timeout* expression is expressed using the symmetric composition of *Select* expression and $Rtimer(10)$ inside an asymmetric composition.

2.2. Formal semantics

Both operational semantics and denotational semantics are defined for Orc. The operational semantics is defined based on transition systems [MC07, WKCM08]. The denotational semantics of Orc language is defined using trees [MHM04] and traces [WKCM08]. In [WKCM08], it is proved that the denotational semantics and operational semantics of Orc are equivalent. In this work, we focus on the operational semantics of the Orc language as explained below.

In [MC07], two types un-timed operational semantics are proposed for Orc: asynchronous operational semantics (AOS) and synchronous operational semantics (SOS). The difference between the two semantics lies on the execution order of the enabled events. Similar to most process algebras, AOS of Orc allows arbitrarily interleaved of all (enabled) events. It does not specify when particular events take place, nor any specific order in processing the events. Hence an enabled event can be arbitrarily delayed by executing other events. In order to program time-out or any other timed-based computation, SOS is introduced so that internal events (i.e., all events other than external response) are processed as soon as possible. External responses are executed if there is no more internal events.

In [WKCM08], AOS is extended to include the time at which an event occurs. This extended semantics is refereed as to timed asynchronous operational semantics (TAOS). The corresponding executions are changed from a sequence of events to a sequence of time-event pairs. This semantics allows multiple events to occur at a single instant of time. An important feature of TAOS is that time can be considered either discrete or continuous. SOS mentioned above is no longer interesting, since the timing aspect of the language is modeled explicitly using timed transitions in [WKCM08].

In this work, we adopt TAOS as defined in [WKCM08] to capture the timing behaviors and verify timing related properties, (since SOS is a temporary solution to support real-time operators like time-out in the un-timed semantics). The semantic model of an Orc program in TAOS is a transition system, which can be generated based on the small-step operational semantics rules. The labels are time-event pairs (t, a). The transition relation $f \overset{t,a}{\to} f'$ states that expression f may take a transition labeled with event a to expression f', such that the transition occurs exactly t time units after its evaluation starts.

Definition 1 Given an Orc program f, the transition system associated with the program is $\mathcal{O}_f = (O, o_0, \mathbb{T} \times \Sigma, \longrightarrow_1)$ where O is the set of possible Orc configurations, o_0 is the initial configuration, $\mathbb{T}$ is the transition time, Σ is the alphabet which includes all events in f, and $\longrightarrow_1 \subseteq O \times (\mathbb{T} \times \Sigma) \times O$ is the transition relation defined by the transition rules in Fig. 1.

In this semantics, we partition the set of events into publication events (written as $!m$) and internal events (written as τ). Publication events correspond to the communication of value m to the environment during a transition. Internal events correspond to the state changes not intended to be observable by the environment. Both publication and internal events are referred to as *base* events. The times in the transition relation are relative to the start of evaluation of the expression. The complete small-step operational semantics of Orc is given in Fig. 1.

The SiteCall rule in Fig. 1 describes the operational semantics of site calls. It specifies that expression $M(m)$, the invocation of site M with value m, performs an internal event at relative time 0 (i.e., without delay) and transitions to an intermediate expression $?u$. We write $\Sigma(M, m)$ for the set of handles that correspond to expression $M(m)$. Each handle describes a possible behavior of site M when it is called with value m. We also call $?u$, the expression corresponding to handle u. Informally, a handle specifies the relative times at which particular values could potentially be returned by a site call, and also the possibility of perpetual non-response. A handle is a set of pairs (t, m), where t is a time and m is a value, denoting that m may be returned at time t as a response. Additionally, a handle may also include a distinguished element ω, which indicates non-response.

The SiteRet rule describes the behavior of handles as a set of potential responses in time. If $(t, m) \in u$, then $?u$ may transit after t units with event $!m$ to $\mathbf{0}$, an expression which has no observable transitions. If $\omega \in u$, then it is possible that the handle will never respond, in which case the call blocks indefinitely. If a handle specifies more than one potential actions (i.e., response or non-response), any one of the values may be returned at the associated time.

Expressions are evaluated using call-by-name in the Def rule, where a single global set of definitions is defined as D. Interested readers can find the detailed explanation of combinator rules (Seq1N, Seq1V, Sym1, Sym2, Asym1, Asym2V and Asym2N) in [WKCM08].

Fundamental sites Fundamental sites have predefined and predictable behaviors. As a result, we can define $\Sigma(M, m)$ completely for a fundamental site M and any value m. In the following definitions, we write $\star$ for signal, a unit value. For the sites in Table 1, there is exactly one handle for each site for a specific parameter value.

$$\begin{array}{ll}\Sigma(\mathbf{0}) = \{\{\omega\}\} & \Sigma(let, m) = \{\{(0, m)\}\} \\ \Sigma(if, true) = \{\{(0, \star)\}\} & \Sigma(if, false) = \{\{\omega\}\} \\ \Sigma(Rtimer, t) = \{\{(t, \star)\}\} & \Sigma(Signal) = \{\{(0, \star)\}\}\end{array}$$

Time-shifted expressions A time-shifted expression, written f^t, is the expression that results from f after t units have elapsed without occurrence of an event. When it is not possible for t time units to elapse without f engaging in an event, we write $f^t = \bot$, where $\bot$ is an unreachable expression. The time-shifted expression f^t, for $t \geq 0$, is defined below based on the structure of f.

$$\begin{array}{ll}M(x)^t & = M(x) \\ M(m)^t & = \begin{cases} M(m) & \text{if t = 0} \\ \bot & \text{otherwise.}\end{cases} \\ E(\bar{p})^t & = \begin{cases} E(\bar{p}) & \text{if t = 0} \\ \bot & \text{otherwise.}\end{cases} \\ (f \mid g)^t & = (f^t \mid g^t) \\ (f >x> g)^t & = (f^t >x> g) \\ (f <x< g)^t & = (f^t <x< g^t)\end{array}$$

3. Timed automata approach

This section presents the Timed Automata based approach. Timed Automata and Uppaal are briefly introduced in Sect. 3.1. Section 3.2 presents an executable model in Timed Automata for each and every constructor in Orc. Section 3.3 demonstrates how Uppaal is used to verify Orc programs using two case studies.

3.1. 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 clocks C, the set of clock constraints $\Phi(C)$ is defined as:

$$\phi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \phi_1 \wedge \phi_2, \text{ where } x \text{ is a clock variable and } c \in \mathbb{R}^+.$$

A clock valuation v for a set of clocks C is a function which assigns a real value to each clock. A clock valuation v satisfies a clock constraint δ, written as $v \models \phi$, if and only if ϕ evaluates to true using the clock values given by v. For $d \in \mathbb{R}_+$, let $v + d$ denote the clock valuation v' such that $v'(c) = v(c) + d$ for all $c \in C$. For $X \subseteq C$, let clock resetting notion $[X \mapsto 0]v$ denote the valuation v' such that $v'(c) = v(c)$ for all $c \in C \setminus X$ and $v'(x) = 0$ for all $x \in X$.

Definition 2 (Timed automaton). 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 a Timed Automaton, a state is associated with an invariant, while a transition is labeled 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 executions 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 $(s, e, \phi, X, s') \in T$ is fired only if ϕ and $I(s)$ are satisfied by the current clock valuation v and $[X \mapsto 0]v$ satisfies $I(s')$. After event e occurs, clocks in X are set to zero. For example, Fig. 2 illustrates some simple Timed Automata. Graphically, a double-lined circle indicates an initial state. Typically, a model of a complex system consists of a network of Timed Automata.[2]

[2] We may treat an automata network as one automaton 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.

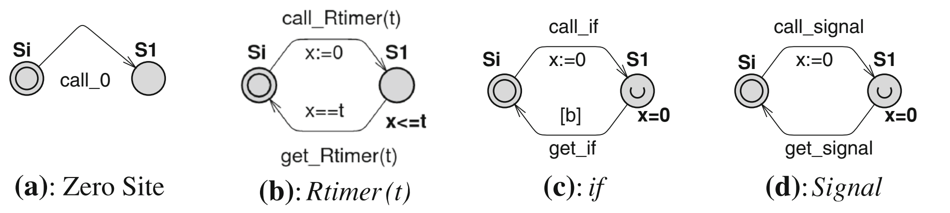


Fig. 2. Fundamental sites

Definition 3 (Timed automata network). Let $\mathcal{A}_i = \langle S^i, s_0^i, \Sigma^i, C^i, I^i, T^i \rangle$ where $i \in \{1, \ldots, n\}$ be a set of Timed Automata. A network of Timed Automata is the parallel composition of $\mathcal{A}_1, \ldots, \mathcal{A}_n$, denoted as $\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$. $\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ is a Timed Automaton $\langle S, s_0, \Sigma, C, I, T \rangle$ such that $S = S^1 \times \ldots \times S^n$; $s_0 = (s_0^1, \ldots, s_0^n)$; $\Sigma = \Sigma^1 \cup \ldots \cup \Sigma^n$; $C = C^1 \cup \ldots \cup C^n$; I is defined such that for all $(s^1, \ldots, s^n) \in S, I((s^1, \ldots, s^n)) = I^1(s^1) \wedge \ldots \wedge I^n(s^n)$; T is the least transition relation which satisfies the following condition.

- For all $(\ldots, s^i, \ldots) \in S$, if $(s^i, e, c, g, q^i) \in T^i$, then $((\ldots, s^i, \ldots), e, c, g, (\ldots, q^i, \ldots) \in S) \in T$.
- For all $(\ldots, s^i, \ldots, s^j, \ldots) \in S$, if $(s^i, e!, c, g, q^i) \in T^i$ and $(s^j, e?, c', g', q^j) \in T^i$, then $((\ldots, s^i, \ldots, s^j, \ldots), \tau, c \cup c', g \wedge g', (\ldots, q^i, \ldots, q^j, \ldots) \in S)$.

Similarly, the semantic model of timed automata is also a transition system as defined below.

Definition 4 Given a Timed Automaton $\mathcal{A} = \langle S, i, \Sigma, C, I, T \rangle$, the transition system associated with the automaton is $\mathcal{T}_{\mathcal{A}} = (\mathcal{S}, s_0, \mathbb{T} \times \Sigma, \longrightarrow_2)$ where $\mathcal{S} = S \times V$ is the set of all possible states. Each state is composed of a control state in S 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. $\longrightarrow_2 \subseteq \mathcal{S} \times (\mathbb{T} \times \Sigma) \times \mathcal{S}$ has the following transitions.

- $\langle s, v \rangle \xrightarrow{\delta, a}_2 \langle s', v' \rangle$ 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 $v'(x) = v(x) + \delta$, for all $x \notin X$. □

Uppaal [LPW97] is our choice of model checker for verifying a network of Timed Automata because of its efficiency as well as its wide recognition. Uppaal is a tool for modeling, simulation and verification of real-time systems modeled 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 means 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 [HNSY92]. In a nutshell, Uppaal is a model checker for systems that can be modeled 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 Web Service orchestration.

3.2. Timed automata semantics for Orc

This section is devoted to a definition of Timed Automata semantics for Orc programs, which allows us to systematically construct a Timed Automata model from an Orc program. 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 [AFP+02], simulation [ADW00], theorem proving [LW00] or more importantly formal verification [LPW97]. In the following, the Timed Automata semantics for Orc programs is formally defined, starting with fundamental sites. The dining philosopher example is used as a running example.

Definition 5 (Zero site). A zero site **0** is modeled as an automaton $\mathcal{A}_{\mathbf{0}}$ where $S = \{s_i, s_1\}$ and[3] $\Sigma = \{call_{\mathbf{0}}\}$ 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 site return event, as illustrated in Fig. 2a. 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] This means that there is no invariant for any state.

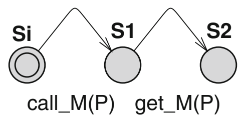


Fig. 3. Timed automaton for site call

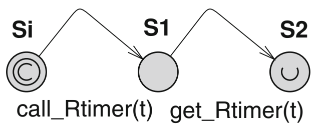


Fig. 4. Timed automaton for *Rtimer*(t) call

Definition 6 (Rtimer(t)). A $Rtimer(t)$ site is modeled as an automaton $\mathcal{A}_{Rtimer(t)}$ where $\Sigma = \{call_{Rtimer(t)}, get_{Rtimer(t)}\}$, $S = \{s_i, s_1\}$, $C = \{x\}$, $I = \{(s_1, x \leq t)\}$ 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 Fig. 2b. 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. This matches the handles set of the *Rtimer* (i.e., $\Sigma(Rtimer, t) = \{\{(t, \star)\}\}$).

Similarly, fundamental sites $call_{if}$ and $call_{Signal}$ are defined as Timed Automata as well, which are illustrated in Fig. 2c, d respectively. The urgent state[4] $S1$ makes sure that the site return event takes no time. $call_{Atimer(t)}$ 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 *get* event without condition b.

The fundamental sites presented so far are defined as the complete expression calls (see Definition 11). 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 7 (Site call). A site call $M(P)$ is modeled as an automaton $\mathcal{A}_{M(P)}$ where $S = \{s_i, s_1, s_2\}$, $\Sigma = \{\tau_{call_{M(P)}}, !m_{get_{M(P)}}\}$, $C = \varnothing$, $I = \varnothing$, and $T = \{(s_i, \tau_{call_{M(P)}}, \varnothing, true, s_1), (s_1, !m_{get_{M(P)}}, \varnothing, true, s_2)\}$. □

A site call is modeled as a Timed Automaton allowing a *call* event which invokes the service and a *get* event which gets the response from the called site, illustrated in Fig. 3. Note that event $\tau_{call_{M(P)}}$ (or $!m_{get_{M(P)}}$) is to distinguish with other τ (or $!m$) events for presentation purpose; and we use $call_{M(P)}$ (or $get_{M(P)}$) in the Timed Automata to denote the $\tau_{call_{M(P)}}$ (or $!m_{get_{M(P)}}$) event. This conforms to the operational semantics of site call, i.e., the two steps of invocation and response as in Fig. 1.

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 Fig. 4 (*Signal* calls are omitted for the similarity). The initial state is set as committed state[5], 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, 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.

External sites must be specified as separate Timed Automata running parallel with the orchestration program for the sake of verification. The behavior of external sites shall be modeled based on their specification, and the invocation and response shall be modeled explicitly and synchronized with the corresponding site call (as specified in Definition 7). For example, the behaviors of the forks in the dining philosopher example are modeled as in Fig. 5, where the user may repeatedly get the fork and then put it back. Consequently, a site call $Fork_i.put$ is the synchronization on the invocation event $call_{Fork_i.put}$ (simplified as *call_fork_i_put* in this example) and response $get_{Fork_i.put}$ event (simplified as *get_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]

[4] 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.

[5] 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.

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

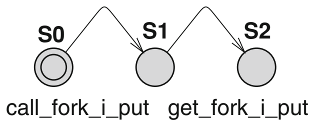
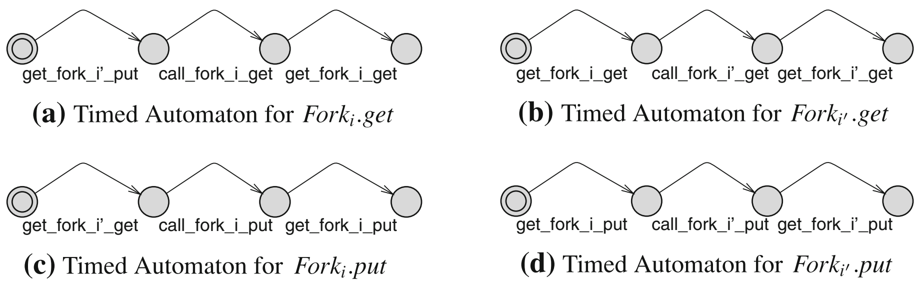


Fig. 5. Timed automaton for $Fork_i$



(a) Timed Automaton for $Fork_i.get$ **(b)** Timed Automaton for $Fork_{i'}.get$ **(c)** Timed Automaton for $Fork_i.put$ **(d)** Timed Automaton for $Fork_{i'}.put$

Fig. 6. Network of automata for $P'_i(1 \leq i \leq N)$

Definition 8 (Sequential composition). Let the automata network of g be $\mathcal{A}_g \mathrel{\widehat=} \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$, where for all $i : 1 \mathinner{..} n$, $\mathcal{A}_i \mathrel{\widehat=} \langle S^i, s^i_0, \Sigma^i, C^i, I^i, T^i\rangle$. A sequential composition $f >x> g$ is modeled as a Timed Automata network $\mathcal{A}_{f>x>g} \mathrel{\widehat=} \mathcal{A}_f \parallel \mathcal{A}'_g$, where $\mathcal{A}'_g \mathrel{\widehat=} (\mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_n)^k$ and for all $i : 1 \mathinner{..} n$, $\mathcal{A}'_i \mathrel{\widehat=} \langle S^{i'}, s^{i'}_0, \Sigma^{i'}, C^{i'}, I^{i'}, T^{i'}\rangle$ where $S^{i'} = S^i \cup \{s^{i'}_0\}$ and $\Sigma^{i'} = \Sigma^i \cup \{get_x\}$ and $C^{i'} = C^i$ and $I^{i'} = I^i$ and $T^{i'} = T^i \cup \{(s^{i'}_0, get_x, \varnothing, true, s^i_0)\}$. □

Notice that a channel[7] named get_x is defined to synchronize the publishing of a value of x from automata $\mathcal{A}_f$ and the receiving of the value in $\mathcal{A}'_g$.

A sequential composition $f >x> g$ is modeled as, in general, a network of Timed Automata $\mathcal{A}_{f>x>g}$, where network of f is untouched, whereas the each automaton $\mathcal{A}_i$ in the network of g is changed by adding a new initial state $s^{i'}_0$ and new transition from $s^{i'}_0$ to s^i_0, which is the original initial state. This new transition has to synchronize on the event get_x, which is to pair with the publishing event from $\mathcal{A}_f$. The purpose of adding this new state and transition is to make sure that if there is a value published from $\mathcal{A}_f$, the execution of g will be triggered. This reflects the SEQ1N and SEQ1V rules in Fig. 1.

In an abuse of 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, Fig. 6 presents the automata interpretation of the $P'_i(1 \leq i \leq N)$ in the dining philosopher example, where each site call is modeled as a Timed Automaton 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 9 (Symmetric parallel composition). A symmetric parallel composition $f \mid g$ is modeled as a network of two Timed Automata (networks) $\mathcal{A}_f \parallel \mathcal{A}_g$. □

A symmetric parallel composition is modeled 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. Note that the clocks in the two automata change at same pace so that the timed shift are satisfied automatically. 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 Fig. 6 (one for each i).

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

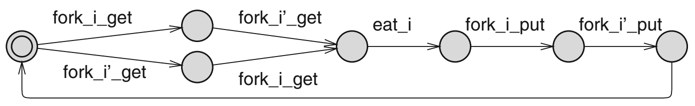


Fig. 7. Timed automaton for P_i

The last compositional constructor of Orc is the asymmetric parallel composition, denoted $f\ <x<\ g$. According to the semantics in [CM05], 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 10 (Asymmetric parallel composition). Let *flag* be a global Boolean variable. It is initially *true*. Let the network of the expression g be $\mathcal{A}_g \mathrel{\widehat=} \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$. An asymmetric parallel composition $f\ <x<\ g$ is modeled as a network of Timed Automata $\mathcal{A}_{f<x<g} \mathrel{\widehat=} \mathcal{A}_f \parallel \mathcal{A}'_g$ where, $\mathcal{A}'_g = \mathcal{A}'_1 \parallel \ldots \parallel \mathcal{A}'_n$ 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

$$\begin{aligned} T = &\{(s_1, get_x, cl', gc, s_2) \mid (s_1, get_x, cl, gc, s_2) \in \mathcal{A}_i.T\} \\ &\cup \{(s_1, e, cl, gc \wedge flag, s_2) \mid e \neq get_x \wedge (s_1, e, cl, gc, s_2) \in \mathcal{A}_i.T\} \end{aligned}$$

where cl' sets flag to false and resets the clocks in *cl usingassignmentin* 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 $publisget_x$ is required. Therefore, it may make steps in parallel or even before g does. Similarly to the symmetric parallel composition, the clocks in $\mathcal{A}_f$ and $\mathcal{A}_g$ change at the same pace so that the timed shift are satisfied automatically. We remark that while our definitions of Timed Automata interpretation for Orc programs are generic, there are plenty of simplifications and optimizations to be performed on the constructed Timed Automata. For example, the P_i expression is modeled (and simplified) as the automaton in Fig. 7.

Definition 11 (Expression call). An expression call is $E(P)$ with $E(P) \mathrel{\widehat=} f$ is modeled as the network of Timed Automata for f prefixed by the $call_E(P)$ event, i.e., $\mathcal{A}_{E(P)} \mathrel{\widehat=} \langle S, s_i, \Sigma, C, I, T\rangle$, where $S = \{s_i \cup \mathcal{A}_f.S\}$ and $\Sigma = \{\tau_{call_{E(P)}} \cup \mathcal{A}_f.\Sigma\}$ and $C = \mathcal{A}_f.C$ and $I = \mathcal{A}_f.I$ and $T = \{(s_i, \tau_{call_{E(P)}}, \varnothing, true, \mathcal{A}_f.s_i) \cup \mathcal{A}_f.T\}$. □

For each parameter x of the expression call, a channel get_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 tail 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 Fig. 7. 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 by focusing only on finite-state Orc programs as discussed below.

In general, our modeling of Orc may end up with a network containing an infinite number of automata (see Definitions 8 and 11). One evidence of a possibly infinite number of automata is that Orc in general allows an irregular language (as in automata theory) as Orc is Turing complete. Some Orc programs that we regard as problematic are as follows.

$$\begin{aligned} &P \mathrel{\widehat=} b \mid (a \gg P \gg c), \text{ where } a, b, c \text{ are sites or even expressions} \\ &M \mathrel{\widehat=} let(x) <x< (let(0) \mid Signal) \\ &N \mathrel{\widehat=} f(x) <x< N \end{aligned}$$

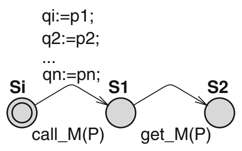


Fig. 8. Timed automaton for site call with value passing

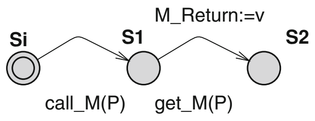


Fig. 9. Timed automaton for site with value passing

P in general allows a language of the form a^nbc^n which is a typical example of an irregular language. It is a known fact that such languages cannot 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. Our target is therefore a subset of Orc language that is regular, type-safe and only allows a finite number of threads. Formally, it is the subset of Orc that are *finite-state* following the definition of finite-state processes given in [OW02]. Intuitively, a finite-state Orc program generates only a finite-state transition system (according to the small-step operational semantics) assuming each step takes one time unit.

The soundness of the Timed Automata modeling is proved by showing that there is a weak bi-simulation relation (denoted using symbol $\approx$) between the Timed Automata and the operational semantics of Orc. The formal definition of $\approx$ is given by Definition 13 in Appendix A. The following theorem is proved by a structural induction over our definitions and the operational semantics of Orc defined in [MC07] (see Appendix A for the proof details).

Theorem 3.1 For any finite-state Orc program f, let $\mathcal{A}_f$ be the timed automata constructed by following Definition 5 to 11. $\mathcal{A}_f$ a weak bi-simulation of f, i.e., $\mathcal{A}_f \approx f$. □

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 globally shared variables as 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 Fig. 8. The return value of a site is assigned in the *get* event in the Site model (Fig. 9). The sequential composition $f\ >x>\ g$ has an additional assignment $x := f_{Return}$ for variable x in the *get* event of f. Similarly for asymmetric parallel composition $f\ <x<\ g$, we add the assignment $x := g_{Return}$ in the *get* event of g. The expression call $E(P) \mathrel{\widehat{=}} f$ has also an assignment in the *get* event for its return value.

3.3. Verification using UPPAAL

This section is devoted to a discussion on how to apply tool support for Timed Automata, in particular UPPAAL, to formally analyze the constructed Timed Automata.

The verification of Orc programs is based on the UPPAAL verifier, which supports a simplified version of timed Computational Tree Logic (CTL). Similar to CTL language, the query language in UPPAAL consists of path formulae and state formulae except the nested path formulae. State formulae describe individual states, whereas path formulae quantify over paths or traces of the model. Path formulae can be further classified into reachability, safety and liveness. Figure 10 illustrates the different path formulae supported by UPPAAL. Each type is described below. Real world examples will be found in Sect. 5.

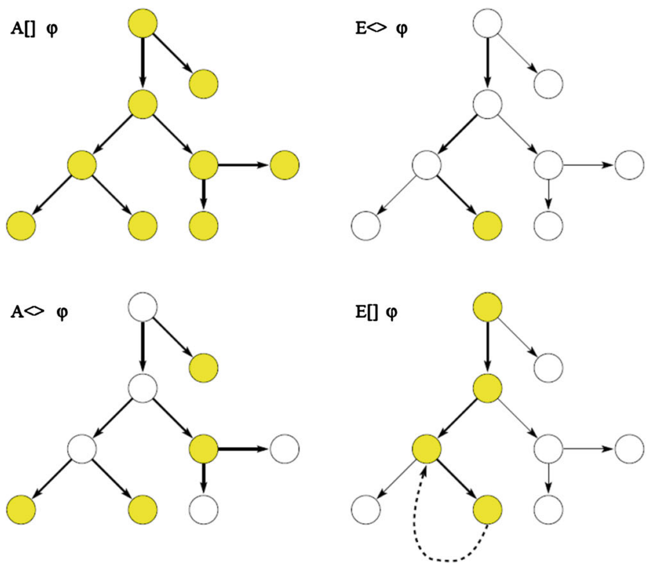


Fig. 10. Path formulae supported in Uppaal. The filled states are those for which a given state formulae φ holds. *Bold* edges are used to show the paths the formulae evaluate on.

State formulae A state formula is an expression that can be evaluated for a state without looking at the behavior of the model. For example, we can check whether $i == 7$ is true at some states. In UPPAAL, deadlock is expressed using a special state formula (although this is not strictly a state formula). The formula simply consists of the keyword deadlock and is satisfied for all deadlock states.

Reachability properties Reachability properties are the simplest form of properties. They ask whether a given state formula, φ, possibly can be satisfied by any reachable state (written as $E\Diamond\ \varphi$). Reachability properties are often used while designing a model to perform sanity checks. For instance, when creating a model of a communication protocol involving a sender and a receiver, it makes sense to ask whether it is possible for the sender to send a message at all or whether a message can possibly be received.

Safety properties Safety properties are on the form: "bad things will never happen". For instance, in a model of computer, a safety property might be, the temperature of the CPU is always under the certain threshold. A variation of this property is that "something will possibly never happen". For instance when playing a game, a safe state is one in which we can still win the game. In UPPAAL, we can use $A\Box\ \varphi$ and $E\Box\ \varphi$ to present the two kinds of properties respectively.

Liveness properties Liveness properties are of the form: something will eventually happen, e.g., when pressing the close button in the lift, then eventually the door will be closed. In its simple form, liveness is expressed with the path formula $A\Diamond\ \varphi$, meaning φ is eventually satisfied. The more useful form is the leads to or response property, written $A\Diamond\ \varphi$ and $\psi \rightsquigarrow \varphi$, which is read as whenever ψ is satisfied, then eventually φ will be satisfied, e.g., whenever a message is sent, then eventually it will be received.

Timing properties Since UPPAAL uses a continuous time model, we can check the clock related prosperities. Clock variables can be used in the properties as propositions. For example, we can check the following timing properties related to a robot.

- $A\Box$ *Robot.move imply* $(x >= 2$ *and* $x <= 3)$ shows that the robot can only move when the clock x is between 2 and 3, i.e., after a delay between 2 and 3 time units.
- $E\Diamond$ *Robot.idle and* $x > 3$: the robot can be idle after 3 time units.

3.4. Automated construction

We developed an experimental tool to automatically construct UPPAAL models from Orc programs 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 Timed Automata according to our definitions in Sect. 3.2. The output of the program is an XML representation of the UPPAAL model, which is ready to be employed and verified. The tool and Orc programs appeared in this paper can be found on the web [DLSZ].

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 *get* 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 8, if we are certain that there is only one copy of g required, we may do the product of the two automata and remove the *get* 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 so as to speed up the verification. However, the simplification and optimization remains as a challenging task and can be further improved 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 program 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: $A\Box$ *not deadlock*. UPPAAL reports that the property does not hold for the system. A counter-example 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 program is deadlock-free and it satisfies properties like that no more than half of the philosophers can be eating at the same time etc.

3.5. Limitations of timed automata approach

The Timed Automata approach gives us an easy way to simulate and verify Orc programs by reusing the existing powerful tools, like UPPAAL. However, as discussed in this section, we can see several limitations of the Timed Automata approach. Firstly, limited by the expressiveness power of Timed Automata, it can only support a subset of Orc language that is regular, type-safe and allows only a finite number of threads. Secondly, the verification of Orc programs relies on the model checker UPPAAL; we can only query a subset of CTL query excluding the nested path formulae. Furthermore, the deadlock state formula can only be used with reachability and invariantly path formulae. Thirdly, Timed Automata model checker is not optimized for the translated Orc program. This is because the mapping from the process algebra definitions of Orc programs to Timed Automata models is not simply one to one. Practical orchestration models often result in high complexity in the corresponding Timed Automata models, which can make the simulation and verification inefficient or even impossible. Finally, to simplify the generated Timed Automata models is difficult. It is possible to identify some patterns in the generated Timed Automata for simplification. However, to systematically identify the patterns is difficult due to the lack of mathematical theory support.

These limitations come from the nature of Timed Automata, so Timed Automata is not the ideal technique for the orchestration verification. We need a complementary solution.

4. Constraint logic programming approach

This section presents our second approach based on Constraint Logic Programming (CLP). Section 4.1 briefly introduces the background of CLP. Section 4.2 illustrates the encoding of Orc programs in CLP. Section 4.3 presents properties we may perform over systems in the CLP framework.

4.1. CLP preliminaries

Constraint Logic Programming (CLP) [JM94] began as a natural merger of two declarative paradigms: constraint solving and logic programming, in which logic programming is extended to include concepts from constraint satisfaction. This combination helps make CLP programs more expressive, and in some cases more efficient than other logic programming languages. The CLP scheme defines a class of languages based upon the paradigm of rule-based constraint programming, where CLP($\mathcal{R}$) [JMSY92] is an instance of this class. In this section, we present some preliminary definitions about CLP. More details can be found in [JM94].

Factorial The following is a typical CLP program:

$fac(0, 1).$
$fac(N, X_1 * N) \vdash N > 0, fac(N - 1, X_1).$

A relation *fac(N, X)* is defined, where X is the factorial of N, denoted as $X = N!$. There are two atoms for the relation *fac(N, X)*, where the first atom is a *fact* and the second one is a *rule*.

The *universe of discourse* $\mathcal{D}$ of our CLP program is a set of terms. Real constants and real variables are both *arithmetic terms*. If t_1, t_2 are arithmetic terms, then so are $(t_1 + t_2)$, $(t_1 - t_2)$ and $(t_1 * t_2)$. A *Constraint* is written using a language of functions and relations. If t_1 and t_2 are arithmetic terms, $t_l = t_2$,$t_l < t_2$ and $t_l \leq t_2$ are all *arithmetic constraints*. If, however, not both the terms t_l and t_2 are arithmetic terms, then only the expression $t_l = t_2$ is a constraint. Both these kinds of constraints will be used in programs, and they form a subset of all the predicates which may appear in a program. Because these constraints have predefined meanings, we shall sometimes emphasize this by calling them primitive constraints (or simply constraints when confusion is unlikely). Constraints are used in two ways, in the basic programming language to describe expressions and conditions, and in user assertions, defined below.

An *atom* is of the form $p(\tilde{t})$, where p is a user defined predicate symbol distinct from $=$, $<$ and $\leq$, and $\tilde{t}$ is a sequence of terms. A *rule* is of the form $A \vdash \tilde{B}, \Psi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint Ψ constitute the *body* of the rule. A *goal* has exactly the same format as the body of the rule, in the form of $? - \tilde{B}, \Psi$. If $\tilde{B}$ is an empty sequence of atoms, we call this a (constrained) *fact*. All goals, rules and facts are terms. A *ground instance* of a constraint, atom and rule is defined in obvious way. A *ground instance* of a constraint is obtained by instantiating variables therein from $\mathcal{D}$. The *ground instances* of a goal G, written $\wr G \int$ is the set of ground atoms obtained by taking all the true ground instances of G and then assembling the ground atoms therein into a set. We write $G_1 \models G_2$ to mean that for all groundings θ of G_1 and G_2, each ground atom in $G_1\theta$ appears in $G_2\theta$.

Let $G = (B_1, \ldots, B_n, \Psi)$ and P denote a goal and program respectively. Let $R = A \vdash C_1, \ldots, C_m, \Psi_1$ denote a rule in P, written so as none of its variables appear in G. Let $A = B$, where A and B are atoms, be shorthand for equations between their corresponding arguments. A *reduct* of G using R is of the form

$$(B_1, \ldots, B_{i-1}, C_1, \ldots, C_m, B_{i+1}, \ldots, B_n, B_i = A \wedge \Psi \wedge \Psi_1)$$

provided $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable. A *derivation sequence* is a possibly infinite sequence of goals $G_0, G_1, \ldots$ where $G_i, i > 0$ is a reduct of G_{i-1}. If there is a last goal G_n with no atoms, notationally $(\Box, \Psi)$ and called a *terminal goal*, we say that the derivation is a *successful* and that the *answer constraint* is Ψ. A derivation is ground if every reduction therein is ground.

Derivation We calculate 3! through the goal $? - fac(3, X)$. The following demonstrates a derivation sequence of the goal with three steps. The constraints in the last step are the termination goal answer $X = 6$.

$$\begin{array}{l}
N = 3,\ fac(N,\ X). \\
\qquad\qquad\qquad\qquad \Downarrow \\
N = 3, N > 0, N - 1 = N_1, X = N * X_1, fac(N_1, X_1). \\
\qquad\qquad\qquad\qquad \Downarrow \\
N = 3, N > 0, N - 1 = N_1, X = N * X_1, \\
N_1 > 0, N_1 - 1 = N_2, X1 = N_1 * X_2, fac(N_2, X_2). \\
\qquad\qquad\qquad\qquad \Downarrow \\
N = 3, N > 0, N - 1 = N_1, X = N * X_1, N_1 > 0, N_1 - 1 = N_2, \\
X1 = N_1 * X_2, N_2 > 0, N_2 - 1 = 0, X_2 = 1.
\end{array}$$

4.2. Orc semantics in CLP

This section is devoted to an encoding of the operational semantics of Orc language in CLP. The practical implication is that we may then use powerful constraint solver like CLP($\mathcal{R}$) [JMSY92] to verify Orc programs. Note that we follow the timed asynchronous operational semantics of Orc as presented in Sect. 2.2.

```
def(dp3, [], para(para(sitecall(pi, [0, 3]), sitecall(pi, [1, 3])), sitecall(pi, [2, 3]))).
def(pi, [I, N], seq(sitecall(toeat, [I, N]), sitecall(pi, [I, N]), _)).
def(toeat, [I, N], where(sitecall(getfork, [I1]),
       where(sitecall(getfork, [I]), sitecall(putTwoForks, [I, N, X, Y]), [X]), [Y]))
       ⊢ plusone(I, I1, N).
def(putTwoForks, [I, N, X, Y], seq(seq(seq(let([X, Y]),
       abscall(eat, []), _), sitecall(putfork, [I]), _), sitecall(putfork, [I1]), _))
       ⊢ plusone(I, I1, N).
plusone(I, I + 1, N) ⊢ I + 1 < N.
plusone(I, 0, N).
```

Fig. 11. Dining philosophers in CLP

4.2.1. Orc models encoding

The initial step of our approach is the syntax encoding of Orc programs in CLP syntax, which can be easily automated by syntax rewriting. A relation of the form *def*(N, P, B) is used to define a definition N with body B and formal parameters list P. Then the different expressions are defined using predicates like *para*(f, g) for $f \mid g$, *seq*(f, g, x) for $f\ >x>\ g$, *where*(f, g, x) for $f\ <x<\ g$, and so on. For instance, Fig. 11 is the syntax encoding of dining philosopher problem of size 3 in CLP, which is a symmetric composition of 3 philosophers, while each is a recursive process named *pi*. Rule *plusone* returns I' given I, i.e., returns $I + 1$ if $I + 1$ is less than N, otherwise returns 0. We use the CLP arbitrary variable "_" for the unimportant variables and input arguments. A description of the relation names appearing in Fig. 11 can be found in the following sections.

4.2.2. Operational semantics

We define *oos* (*O*rc *O*perational *S*emantics) relation as a transition system interpretation of an Orc program, where the state is identified by the combination of the process expression and the valuation of the time variables. The relation *oos(P1,T1,E,P2,T2)* is true if the process *P1* may evolve to *P2* through event transition *E*. $T1$ is the time before the transition and $T2$ is the time after the transition. $T1 == T2$ implies that the transition takes no time. *oos* is defined in terms of each and every operator of Orc language. For instance, the site call evaluation steps are defined through the following clauses:

```
oos(sitecall(M, m), T, [tau], u(M, m), T) ⊢ not(def(M, _, _)), constant_list(m).
oos(u(M, m), T1, [get(V)], zerosite, T2) ⊢ T2 >= T1, constant_list(V).
oos(abscall(M, _), T, [get([signal])], zerosite, T).
```

The first rule reflects the SiteCall rule in Orc's semantics. A site call $M(m)$, where M is the site name and m is the list of actual parameters, transitions to $u(M, m)$ with *tau* event. The intermediate state $u(M, m)$ represents a process that is blocked waiting for the return from the call. There are two conditions for this rule to fire: the site call M is not defined internally (i.e., *not*(*def*(M, _, _))) and the input actual parameters must be constant (i.e., *constant_list*(m)). We assume this step takes 0 time unit. Keyword *not* in the rules means "negation as failure" as in logic programming.

The second rule maps to the SiteRet rule in Orc's semantics. The pending process $u(M, m)$ receives a result V and transitions to *zerosite* via a *get*(V) event. The returned time is stored in $T2$. If the environment never produces a response event, then the call blocks infinitely. Therefore, in the modeling process, developers need to provide return values and exact time delay for all returning site calls. The published tuple V must contain only constants, otherwise the transition will be blocked.

The last rule *abscall* is a simplification of the two-step evaluation of the site call. Sometimes the return value and evaluation time are not important for the modeling, we can use abstract site calls to model the two-step execution.

The following rules define the encoding of Orc fundamental sites. The *zerosite* will not progress anymore and leave the timer to keep ticking. The *let*(C) transitions to *zerosite* and generates a *get*(C) event. The *if* rule will return a signal if the given condition C is evaluated to be true,[8] and it is blocked otherwise. The *signal* rule will

[8] The evaluation of the condition C is based on the CLP($\mathcal{R}$) function *call*, which returns true if the input constraint is true.

just publish a signal and reach *zerosite*. The *clock* rule will publish the current time T and reach *zerosite*. We define *rtimer* as a site transiting to *zerosite* with D time unit delay.

$oos(zerosite, T1, [], zerosite, T2) \vdash D > 0, T2 = T1 + D.$
$oos(let(C), T, [get(C)], zerosite, T) \vdash constant\_list(C).$
$oos(if(C), T, [get([signal])], zerosite, T) \vdash call(C).$
$oos(signal, T, [get([signal])], zerosite, T).$
$oos(clock, T, [get([T])], zerosite, T).$
$oos(u(rtimer, [D]), T1, [get([signal])], zerosite, T1 + D) \vdash D >= 0.$

The following rules define the composition operators. The first two rules correspond to Sym1 and Sym2 steps for the symmetric composition. The third and fourth rules correspond to Seq1N and Seq1V steps for the sequential composition. For asymmetric parallel composition, we use rule five and six for Asym2N and Asym2V steps, which allow transitions on processes F and G, but only if the process F expression does not publish a value. When the process F publishes a value, we use rule seven for Asym1 step to terminate F and the result value is bound into variable X and passed into the process G. The last rule defines the expression definition step Def of the Orc semantics. It returns the body of the definition. This rule requires that the actual parameters passed in must be constants.

$oos(para(F, G), T1, [E], para(F1, G1), T2) \vdash oos(F, T1, [E], F1, T2), tshift(G, G1, T2 - T1).$
$oos(para(F, G), T1, [E], para(F1, G1), T2) \vdash oos(G, T1, [E], G1, T2), tshift(F, F1, T2 - T1).$
$oos(seq(F, G, X), T1, [E], seq(F1, G, X), T2) \vdash oos(F, T1, [E], F1, T2), not(E = get(\_)).$
$oos(seq(F, G, X), T, [tau], para(seq(F1, G, X), G), T) \vdash oos(F, T, [get(X)], F1, T).$
$oos(where(F, G, X), T1, [E], where(F1, G, X), T2) \vdash$
$\quad oos(G, T1, [E], G1, T2), not(E = get(\_)), tshift(F, F1, T2 - T1).$
$oos(where(F, G, X), T, [tau], F1, T) \vdash oos(G, T, [get(X)], G1, T), tshift(F, F1, T2 - T1).$
$oos(where(F, G, X), T1, [E], where(F1, G1, X), T2) \vdash oos(F, T1, [E], F1, T2), tshift(G, G1, T2 - T1).$
$oos(sitecall(E, P), T, [tau], F, T) \vdash constant\_list(P), def(E, P, F).$

The parameter passing and variable substitution are solved by the variable unification provided by the CLP language. In the expression definition rule, the actual parameter list P is passed to the *def* rule, which achieves the variable substitution automatically after the unification of the actual parameters P and the formal parameters in the definition. The variable passing of the sequential composition and asymmetric composition also uses the unification of the middle variable X in the definition.

The time shift rules are defined as follows according to the definition in Sect. 2.2. Because the parameter passing are done via the unification, the parameters of a site call cannot be variables. Hence the rule on $M(x)$ is ignored. *bottom* is a constant for invalid state.

$tshift(sitecall(M, C), sitecall(M, C), 0).$
$tshift(sitecall(M, C), bottom, \_).$
$tshift(sitecall(E, P), sitecall(E, P), 0).$
$tshift(sitecall(E, P), bottom, \_).$
$tshift(para(F, G), para(F1, G1), T) \vdash tshift(F, F1, T), tshift(G, G1, T).$
$tshift(seq(F, G, X), seq(F1, G, X), T) \vdash tshift(F, F1, T).$
$tshift(where(F, G, X), where(F1, G1, X), T) \vdash tshift(F, F1, T), tshift(G, G1, T)$

4.2.3. Semantic model

Given an arbitrary CLP program, its execution starts with the initial set of terms and then repeatedly tries to apply rules until the goal or a fix point is reached. In this work, we focus on the encoded CLP program P_f of a given Orc program f. In the following, we formalize the semantic model of P_f to a transition system so that we can argue the equivalence with original Orc program f.

Definition 12 Given an Orc program f, let P_f be the corresponding CLP program constructed using the rules above together with *oos* and *tshift* rules. The transition system associated with the program P_f is $\mathcal{P}_f = (C, c_0, \mathbb{T} \times \Sigma, \longrightarrow_3)$ where C is the set of possible sets of terms, c_0 is the initial set of terms, $\mathbb{T}$ is the transition time, Σ is the alphabet which includes all events in f, and $\longrightarrow_3 \subseteq C \times (\mathbb{T} \times \Sigma) \times C$ is the transition relation defined by the *oos* transition rules.

Note that the only transitions caused in $\mathcal{P}$ is the execution of *oos* rules, i.e., for any execution of *oos(P1,T1,E,P2,T2)*, there is a transition in $\mathcal{P}$ such that $c \xrightarrow{T2-T1,E}_3 c'$. Other rule execution will not generate any transition in $\mathcal{P}$.

Theorem 4.1 For any finite-state Orc program f, P_f a weak bi-simulation of f, i.e., $P_f \approx f$. □

Proof sketch: The theorem can be proved by a structural induction on the Orc programs based on the operational semantic rules as shown in Fig. 1.

Firstly, orc operational semantics rules are triggered by the matching of pre-conditions. Execution of the rules will generate the resulting transition. The execution of the *oos* rules follows the exactly same way.

Secondly, it is clear that the mapping from the operational semantics rules to CLP *oos* rules is strictly one to one as shown in the section above. This implies that whenever there is a transition in $\mathcal{O}_f$, then there is a corresponding transition in $\mathcal{P}_f$, and vice versa. Therefore we can conclude that P_f a weak bi-simulation of f. □

4.2.4. Simplification based on algebra laws

Process algebra can use its algebraic laws to facilitate reasoning and simplification about process definitions. This can bring us significant simplification of Orc programs, which is not available in the Timed Automata approach. We define the following rules according to the equivalence relations of laws of Orc [MC07, LZH10], which are then used to simplify the expressions during the verification.

The following four rules show the simplification laws involving *zerosite*. The first argument of rule *sim* is the input process. The second argument is the simplified process. Cut operator ! terminates the search after the first match, which is used to improve efficiency.

$sim(para(zerosite, P), P) \vdash !.$
$sim(para(P, zerosite), P) \vdash !.$
$sim(seq(zerosite, \_, \_), zerosite) \vdash !.$
$sim(where(F, zerosite, \_), F) \vdash !.$

The following rules define a depth first search to apply a simplification step for a process whenever possible. The simplified process is stored in the second argument of *sim* rule.

$sim(seq(P, Q, [X]), seq(P1, Q, [X])) \vdash sim(P, P1).$
$sim(seq(P, Q, [X]), seq(P, Q1, [X])) \vdash sim(Q, Q1).$
$sim(para(P, Q), para(P1, Q)) \vdash sim(P, P1).$
$sim(para(P, Q), para(P, Q1)) \vdash sim(Q, Q1).$
$sim(where(P, Q, [X]), where(P, Q1, [X])) \vdash sim(P, P1).$
$sim(where(P, Q, [X]), where(P, Q1, [X])) \vdash sim(Q, Q1).$

To simplify an Orc program, we just need to invoke rule *simplify*, which will try to apply the simplification laws until no more laws can be applied. The simplified process is stored in Q.

$simplify(P, Q) \vdash sim(P, D), !, simplify(D, Q).$
$simplify(P, P).$

We do not define all the algebra laws for Orc program here, e.g., commutativity laws for symmetric composition and distributive laws over $f\ {<}x{<}\ g$. The reason is that these laws will not help in the simplification of the expression. However, if needed, we can define them easily in CLP syntax.

We argue that this simplification will not affect the soundness of Theorem 4.1. This is because the simplification rules will not generate any new transitions since no new *oos* is introduced. These rules are just performing the syntax rewriting based on the algebra laws of Orc language, which has been proved in [MC07, LZH10]. Notice that using of cut operator in the simplification rules will not affect the soundness of Theorem 4.1. Because cut operator is to reduce the searching space after the first match of a rule, which means that the simplification may not be performed for all Orc programs. However, as long as the simplification is sound, then Theorem 4.1 is sound.

4.3. Properties verification of Orc models

This section is devoted to properties verification we may perform over Orc programs encoded in CLP. We implemented a model checker prototype in one of the CLP solver, namely CLP($\mathcal{R}$) for its support of real-type variables. Assertions can be proved against a given real-time system. We also developed a number of shortcuts for easy querying and proving.

Using CLP, we may make explicit assertion which is either a safety assertion, or a liveness assertion. Yet it can be used for both purposes using a unique interpretation. A discussion on how to allow such temporal properties is presented in [CCO+04]. In the following, we show how safety properties and liveness properties can be queried. We employ the *coinductive tabling* technique [JJV05] to obtain termination when dealing with recursions, which facilitates verifying safety and liveness properties based on traces. Essentially, *coinductive tabling* extends CLP so that it can inductively use proof obligations that are assumed but not yet proven, and it can generate new proof obligations assertions dynamically. This technique is akin to the notion of tabling in logic programming systems in that the main purpose is to obtain termination when dealing with recursion. In standard tabling, procedure calls and their answers are tabled so as not to repeat them. In our approach, the main differences are first that the setting is CLP and not just logic programming, and more importantly, that proof obligations, procedure calls and their answers are all tabled. Termination is obtained by applying a principle of coinduction, that is: a recursive proof obligation may be proved by assuming that a preceding proof obligation is true.

First of all, the reachability testing can be defined in CLP as follows. The relation *treachable(P, Q, N, T1, T2)* states that it is possible to reach the process expression Q at time $T2$ from P at time $T1$, with trace N. We also apply the simplification rule *simplify* after every step to speed up the operations. By using the tabling method, we dynamically record the process expressions that have been explored in order to avoid re-exploring them. In this regard, reachability can be asserted using *treachable*.

$treachable(P, P, [], T1, T1).$
$treachable(P, Q, [E \mid N], T1, T2) \vdash oos(P, T1, E, P1, T3), simplify(P1, P2), treachable(P2, Q, N, T4, T2).$

An invariant property (a predicate over time variable and state variables and possible local clocks) is in general expressed as the assertion:

$inv(P, T, Property) \vdash not(treachable(P, Q, \_, T, T1), not\ sat(Property)).$

where *not sat(Property)* is a constraint indicating that the output from the previous atom not satisfying the user defined *Property*.

One safety property of special interest is deadlock-freeness. The following clauses are used to prove it.

$tdeadlock(P, T1) \vdash treachable(P, P1, N, T1, T2),$
$\quad (not(oos(P1, T2, [], Q, T), oos(Q, T, [\_], \_, \_)); (oos(P1, T2, [], Q, \_); not(oos, P1, T2, [\_], \_, \_))),$
$\quad printf(\text{“}deadlock\ at : \%\text{”}, [N]).$

Basically, it states that a process P at time *T1* may result in deadlock if it can reach the process expression *P1* at time *T2* where no event transition is available neither at *T2* nor at any later moment. The last line outputs the deadlock trace as a counterexample. Alternatively, we may present it as a result of the deadlock-freeness proving.

We allow trace-based properties (safety or liveness) that can be checked by exploring trace set partially. The retrieve of a trace is done by the predicate $superstep(P, N, Q)$, which finds a sequence of events through which process expression P evolves to Q:

$superstep(P, [\_], \_) \vdash not(oos(P, \_, \_, Q, \_), not\ table(Q)).$
$superstep(P, [A \mid N], Q) \vdash oos(P, \_, M, P1, \_), not(M == []; M == [tau]),$
$\quad M = [A], not\ table(P1), assert(table(P1)), superstep(P1, N, Q).$
$superstep(P, N, Q) \vdash oos(P, \_, M, P1, \_), (M == []; M == [tau]),$
$\quad not\ table(P1), assert(table(P1)), superstep(P1, N, Q).$

We may prove that some event will always eventually be ready to be engaged using the following rule: where rule $member(N, E)$ returns true if event E appears at least once in the event sequence N.

$finally(P, E) \vdash not(superstep(P, N, \_), not\ member(N, E)).$

Rule $finally(P, E)$ captures the idea that there is no such trace without event E in this process P. In other words, this process will eventually go to event E. Another property based on traces would be identifying the relationship among events, e.g., event A can never happen before (after) event B in a trace or trace fragment. Taking the dining philosopher for example, we would like to ensure that in a round of eating, the event $Fork_i.put$ will never happen before event eat_i.

Verification For the dining philosophers, we would like to check that it is deadlock-free by running the following goal and expecting failure:

$? - tdeadlock(dp3, 0)$

For the vending machine example, we would expect that whenever we choose *tea*, it would never dispatch *coffee* instead of *tea*, which can be checked by the following goal, where $in(tea, N)$ means element *tea* is inside the trace N and $after(N, dispatchcoffee, tea)$ means that *dispatchcoffee* appears after *tea*.

$? - superstep(vending, N, \_), not(in(tea, N), after(N, dispatchcoffee, tea)).$

Additional checking In reality, most processes are non-terminating, so it would not be possible to retrieve all possible traces of a process. However, by given a specific trace of a trace fragment, we are able to identify whether it is an event sequencing of a given process. For instance, the following clause is used to query if a sequence of event is a trace of the system, where P is a process expression and X is a sequence of events.

$trace(P, X) \vdash superstep(P, X, \_).$

In addition to proving pre-specified assertions, one distinguished feature of our approach is that implicit assertions may be proved. For example, we may identify the lower or upper bound of a (time or data) variable, which is very useful to do the worst or best case analysis of orchestration plans.

$dur(P, Q, T1, T2) \vdash oos(P, T1, \_, Q, T2).$
$dur(P, Q, T1, T2) \vdash oos(P, T1, \_, P1, T3), dur(P1, Q, T3, T2).$

We are able to compute the duration of the execution of one process P to its subsequent process Q by the above two rules, where T_1 is the starting time and T_2 is the ending time. By using the predicate *dur*, we are able to get identify the lower bound of some processes involving time. If process $Rtimer(5) \gg Google(Orc)$ returns, then it should take more than 5 time units. This can be checked by the following goal and expecting $T \geq 5$.

$? - dur(seq(sitecall(rtimer, [5]), sitecall(Google, [Orc]), \_), zerosite, 0, T) \vdash T >= 5.$

5. Case study and experiments

This section presents a case study of an auction orchestration modeled using both Timed Automata and CLP. The comparison of the two approaches can reveal the difference of the two approaches. We also conducted some experiments to compare the performance of the two approaches.

5.1. Orchestrating an auction

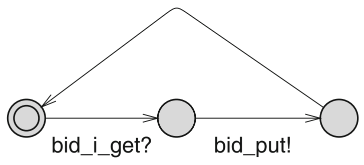
In this subsection, we demonstrate a typical web-based application, i.e., running an auction for an item. This example was originally presented in [MC07].

First, an item can be 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 general, there are multiple *Bidder*s. A *Multiplexor* is used to merge all the bids into a single channel, i.e., *bid*. The Orc definition for Multiplexor is described as follows.

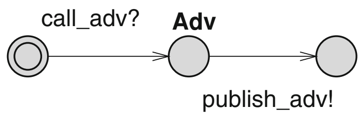
$$Multiplexor_i \mathrel{\widehat{=}} bid_i.get >y> bid.put(y) \gg Multiplexor_i$$
$$Multiplexor \mathrel{\widehat{=}} Multiplexor_1 \mid Multiplexor_2 \mid \ldots \mid Multiplexor_i$$

In UPPAAL, a template called *Bidder* is built, which outputs a bid on channel *bid*. In general, there are multiple *Bidder*s. The Timed Automata for the basic sites are shown in the Fig. 12.

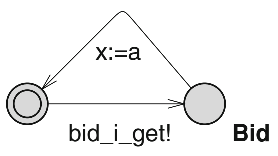
In CLP, the representation of *Multiplexor* is defined straightforwardly as follows, where the structure *ii* is a syntactic sugar for indexed interleaving. *counter_value* is a build-in facility in CLP($\mathcal{R}$) to check whether the value of global variable *iicount* is same as I. *iicount* is updated in *ii* to synchronize with *Multiplexor*.



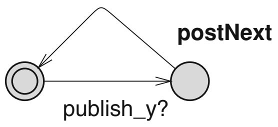
(a): Timed Automaton for *Multiplexor*$_i$



(b): *Adv* site call

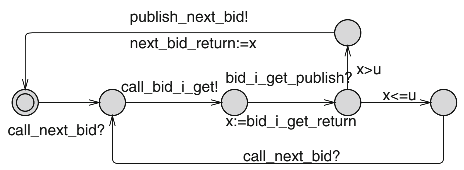


(c): *Bidder*$_i$

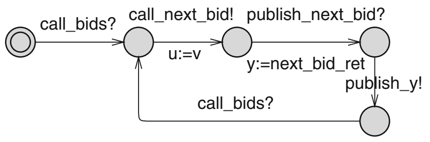


(d): *PostNext* site call

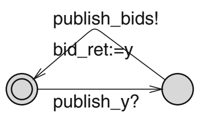
Fig. 12. Basic sites in auction example



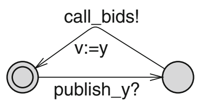
(a): Timed Automaton for *nextBid*(u)



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



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



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

Fig. 13. Basic sites in auction example

$def(multiplexori, [], seq(seq(sitecall(getbid, [I]), sitecall(putbit, [Y]), [Y]),$
$\quad sitecall(multiplexori, [I])), \_) \vdash counter\_value(iicount, I).$
$def(multiplexor, [N], ii(N, sitecall(multiplexori, []))).$

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

5.1.1. 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*.

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

Orc program $nextBid(v)$ returns the next bid from c exceeding v. The site call $if(x > u)$ returns a signal if $x > u$ 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.

$Auction_1(p) \mathrel{\widehat{=}} Adv(p) \gg Bids(p) > z > PostNext(z) \gg \mathbf{0}$

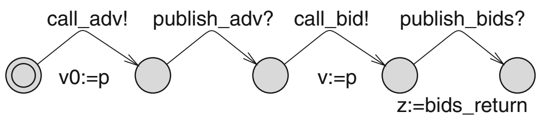


Fig. 14. Timed automaton for *Auction*$_1$ part 1

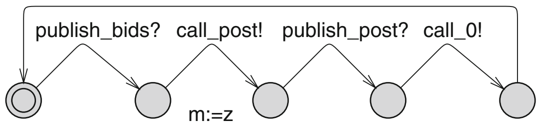


Fig. 15. Timed automaton for *Auction*$_1$ part 2

The Timed Automata of $nextBid(u)$ and $Bids(v)$ are shown in Fig. 13. The Timed Automaton of $nextBid(u)$ is simplified by combining the two if-condition automata with the main $nextBid(u)$ timed Automaton, because the two conditions $(x > u)$ and $(x \leq u)$ are exclusive.

Following the Timed Automata semantics defined in Sect. 3.2, $Auction_1(v)$ is interpreted as the automata in Figs. 14 and 15.

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 $let(x, y, z)$.

The CLP modeling for $Auction_1$ is defined as follows, which is a direct translation.

$def(nextbid, [U], seq(sitecall(bidget, []), para(seq(if(X > U), let([X]), \_),$
$\quad seq(if(U <= X), sitecall(nextbid, [U]), \_)), [X])).$
$def(bids, [V], seq(sitecall(nextbid, [V]), para(let([Y]), sitecall(bids, [Y])), [Y])).$
$def(auction1, [P], seq(seq(seq(abscall(adv, \_), sitecall(bids, [P]), \_),$
$\quad sitecall(postnext, [Z]), [Z]), zerosite, \_)).$

By checking with UPPAAL and CLP($\mathcal{R}$), 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 posted bid is never lower than the minimum acceptable bid (see the second property in Table 3).

5.1.2. Terminating auction

The previous program is modified so that the auction terminates if no higher bid arrives for h time units (say, h hours). 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 h time units.

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

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 [DHQ$^+$04]. For example in Fig. 16, we can use the 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 Fig. 16, in which c denotes the clock and h is a constant.

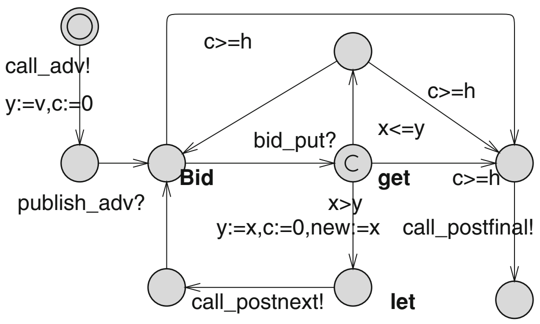


Fig. 16. Auction$_2$: terminating auction

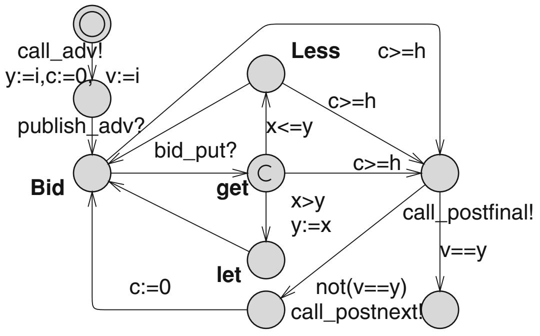


Fig. 17. Auction$_3$: batch processing

The CLP modeling for the *Tbids* and $Auction_2(v)$ is defined as follows, which is a direct translation.

def(*tbids*, [*V*], *where*(
  para(*seq*(*sitecall*(*nextbid*, [*V*]), *let*([*Y*, *true*]), [*Y*]), *seq*(*sitecall*(*rtimer*, [*H*]), *let*([*V*, *false*]), _)),
  para(*let*([*X*, *FLAG*]), *seq*(*if*(*FLAG*), *sitecall*(*tbids*, [*X*]), _)), [*X*, *FLAG*])).
def(*auction*2, [*V*], *seq*(*seq*(*abscall*(*adv*, [*V*]), *sitecall*(*tbids*, [*V*]), _),
    para(*seq*(*seq*(*if*(*FLAG*), *abssite*(*postnext*, *X*), _), *zerosite*, _),
      seq(*seq*(*if*(*FLAG*), *abssite*(*postfinal*, *X*), _), *zerosite*, _)), [*X*, *FLAG*])).

5.1.3. 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 program and the construction. The detail of the auction is available elsewhere in [MC07]. The constructed Timed Automaton is presented in Fig. 17. The CLP model is omitted for its direct translation.

5.2. Experiments

In order to compare the performance of the two approaches proposed, we conducted several experiments on the dining philosopher and auction examples on verifying various properties. The results are presented in this section.

Table 2. Dining philosopher experiment results

Orc	Property	Result	UPPAAL time(s)	CLP sim time(s)	CLP time(s)
Philosopher3	Deadlock	True	1	0.03	0.02
Philosopher4	Deadlock	True	6	1.4	1.4
Philosopher5	Deadlock	True	145	55	53
Philosopher3	Always eat	True	0.01	0.01	0.01
Philosopher4	Always eat	True	0.015	0.012	0.012
Philosopher5	Always eat	True	0.03	0.015	0.015
Philosopher3	Majority eat	False	0.65	0.42	0.42
Philosopher4	Majority eat	False	18	3.3	3.3
Philosopher5	Majority eat	False	930	254	250

Table 3. Experiment results

Orc	Property	Result	UPPAAL time(s)	CLP sim time(s)	CLP time(s)	Remark
$Auction_1$	*A[ ] not deadlock*	True	20	0.1	0.1	Non-terminating.
$Auction_1$	*A[ ] not (PostNext.posted<250)*	True	3	0.1	0.1	No bid price lower 250.
$Auction_1$	*A[ ] not(old==0) imply new>old*	True	90	0.1	0.1	Price posted on the *PostNext* site keeps increasing.
$Auction_1$	*E<> PostNext.posted == 500*	True	1	0.2	0.3	Possible to post 500.
$Auction_2$	*A[ ] not deadlock*	False	1	0.15	0.14	Terminating.
$Auction_2$	*A[ ] PostFinal.postFinal imply Auc.c>=h*	True	150	1.2	1.6	Auction terminates after *h* time units.
$Auction_2$	*A[ ] PostFinal.final == 1000 imply Bidder10.bid == true*	True	10	0.5	0.7	The final bid comes from the respective bidder.
$Auction_3$	*A[ ] not deadlock*	False	1	0.2	0.3	Terminating.
$Auction_3$	*E[ ] not(PostNext.p1<h and PostNext.p1>0)*	True	60	3.6	4.5	It is not possible to post a highest bid before *h* time units.

5.2.1. *Experiment bed*

All experiments were conducted on a Pentium 3.0 GHz processor PC with 1 GB RAM and a 20 GB quota of disk space, running Windows XP. For the Timed Automata approach, we used UPPAAL version 4.0 [LPW97]. The CLP approach is based on a CLP($\mathcal{R}$) program [JMSY92]. Column **CLP Sim** and **CLP** shows the verification time with the simplification rules and without the simplification rules respectively.

5.2.2. *Dining philosopher*

The first bench of experiments is on the Dining Philosophers examples. We implemented this example with N philosophers and N forks. The following three properties are checked. We also tried different numbers of Dining Philosophers to test the verification capability of the two approaches. The result of the running time is shown as shown in Table 2.

Deadlock The system can deadlock.

Always eat It is possible that one philosopher eats all the time with the others starving. This property is checked with trace refinement.

Majority eat Not more than or equal to $(N + 1)/2$ philosophers can eat at the same time.

5.2.3. *Auction*

In the verification experiment of auction example, we created 10 Bidders whose bid prices are from 200 to 1100, while the minimum bid price is 250. Some properties concerning all three auction strategies together the verification time are illustrated in Table 3.

The discussion of the experiments are shown in Sect. 6.

6. Discussion

In this section, we compare the two approaches presented in this work in the following aspects.

Translation vs. encoding The two approaches presented in this work represent Orc programs using different formalisms. The Timed Automata approach performs a systematic translation from Orc programs to Timed Automata based on the operational semantics. The soundness of this approach is proved by showing a bisimulation of translated TA models with the original Orc program. However this translation is not complete, i.e., limited by the expressiveness power of Timed Automata, it can only support a subset of Orc language that is regular, type-safe and allows only a finite number of threads.

The CLP-based approach creates a framework which can interpret Orc programs by encoding them using CLP. This framework includes the operational semantics of Orc languages as well we simplification rules, which allow the encoded Orc programs to be executed by following the operational semantics. The soundness of this approach is proved by showing a bisimulation of the execution model of the encoded CLP programs with the original Orc programs. For the CLP encoding, the Orc programs can be mapped exactly to CLP programs because the encoding is basically a syntax rewriting. Therefore the completeness is guaranteed in this approach.

It is clear now that the two approaches are quite different. The translation based approach may not be complete due to the restricted expressiveness of the targeting formalism. The encoding based approach requires the effort to develop an execution model based on the semantics, which is more complicated, but flexible. One example of flexibility is the application of simplification rules. In the TA approach, the simplification can be done just at the translation step. After the translation, the simplification cannot be applied because Orc program structure information is gone. However, in the CLP approach, the direct encoding keeps the Orc program structure, which makes simplification rules can be applied at any step.

Simulation and verification capability For the TA approach, the simulation and verification of Orc programs rely on tool support for TA, e.g., the model checker UPPAAL. For the verification capability, we can query a subset of *timed CTL* query excluding the nested path formulae, as supported by UPPAAL. For the second approach, the simulation and verification of Orc programs needs to be developed manually. For example, currently, we support safety properties (including deadlock, reachability, invariant) and liveness properties (including eventually operator) in the second approach. Furthermore, new model checking algorithms can be developed in this framework. Note that timed properties are not supported in the CLP-based approach.

UPPAAL is not optimized for the translated Orc programs. This is because the mapping from the process algebra definitions of Orc programs to Timed Automata models is not simply one to one. Practical orchestration models often result in high complexity in the corresponding Timed Automata models, which can make the simulation and verification inefficient or even impossible. For the CLP approach, there is no such problem. However, optimizations techniques (e.g., partial order reduction or symmetry reduction) need to be realized manually.

When the verification fails, there will be a counterexample generated by both approaches. The counterexample in UPPAAL is presented as a sequence of events, which can be simulated using UPPAAL simulator. Since there is a mapping between original Orc program and generated Timed automata (mainly the invocation and response of the side calls), we can trace the counterexample in the original Orc program. But this may not be always easy. For example, in a case that two or more events with same name are enabled, if the counterexample trace include one such event, to find out the correct one needs careful examination of the whole trace. In the CLP-based approach, the counterexample is stored in a list (e.g., the third item in the *treachable* predicate). The interpretation of the counterexample in the CLP-based approach is straightforward since the Orc encoding of the program reflects exactly the original Orc program.

Performance and scalability From the experiments presented in Tables 2 and 3 of Sect. 5, it can be noticed that the performance of the CLP-based approach is much more scalable than UPPAAL. The reason is that CLP-based approach introduces no overhead during the encoding process and the resulted state space is smaller than the TA approach. One typical cause of the overhead transition of the TA approach is the additional transition introduced in the sequential composition. However, the CLP-based approach is still suffering from the state-space explosion problem, e.g., the deadlock and majority eat properties in dining philosophers example. The efficiency of the simplification in the CLP approach heavily depends on the models. In dining philosophers example, there is no performance gain. The overhead of the simplification checking is negligible. The simplification rules improve the performance for the actions examples.

From Table 3, we can see that given the same model, the time for verifying different properties are similar for CLP-based approach. For UPPAAL, if the property is false, the verification time can be very short. This reflects that the on-the-fly model checking approach in UPPAAL will stop immediately when a counterexample is found. For the CLP-based approach, the termination of the rule execution is hard to predict due to using of tabling and other optimization technique in the CLP($\mathcal{R}$) tool.

Overall, the CLP-based approach is more scalable than the TA approach based on the experimental results. However, if the properties involve time, only UPPAAL tool can provide such support.

7. Conclusion and future works

In this work, we presented two promising approaches which can simulate and verify Orc programs. In the first approach, an automata-based semantics for Orc language is proposed, which allows a systematic construction of Timed Automata models from Orc programs. After that, we explored ways of using UPPAAL to verify critical properties over Orc programs. We developed a tool to automate our approach. In the second approach, we encode the semantics of Orc in CLP, which allows us to verify the properties using powerful CLP tools. Because the one-to-one mapping of the encoding, CLP encoding can support the complete Orc language. Furthermore, a wider range of properties can be verify in the open modeling environment. We conducted some experiments to compare the two approaches and found CLP approach is fast and more expressive.

There are some possible future works. Starting from the first approach, one 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. A more direct approach could be to develop a tool to support the verification of Orc language itself without any translation. This is possible because Orc language has well defined operational semantics and its behaviors can be interpreted as labeled transition systems. If the behaviors have finite states, automatic verification techniques like model checking can be used to verify the Orc programs. In the CLP approach, clauses are defined to represent the operational semantics of Orc. Because of the natural connections among operational semantics, algebra semantics and denotational semantics, it is interesting to look at the different semantics by encoding them using CLP.

Starting from the second approach, we want to continue the CLP approach further to look at its verification ability for process algebra. Secondly, we plan to investigate the possibility to verify the state-of-the-art Web Service orchestration language WS-BPEL using CLP, which has no formal semantics, but much more complicated syntax. Finally, we want to extend our work to the Web Service Choreography, which is a multi-party contract that describes from global view point the external observable behavior across multiple Web Services. The long term objective is to develop a generic modeling and verification framework for SOC.

A. 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 programs refer to a subset of Orc language that is regular, type-safe and with a finite number of threads (see Sect. 3.3 for details).

Definition 13 Let $\mathcal{O}_1 = (C, c_0, \mathbb{T} \times \Sigma, \longrightarrow_1)$ and $\mathcal{O}_2 = (S, s_0, \mathbb{T} \times \Sigma, \longrightarrow_2)$ be two transition systems, For any $c \in C$ and $s \in S$, $c \approx s$ if and only if,

- $\forall\, t \in \mathbb{T}, \alpha \in \Sigma, c \xrightarrow{t,\alpha}_1 c'$ implies there exists $s' \in S$ such that $s \xrightarrow{t,\alpha}_2 s'$, and $c' \approx s'$.
- $\forall\, t \in \mathbb{T}, \alpha \in \Sigma, s \xrightarrow{t,\alpha}_2 s'$ implies there exists $c' \in C$ such that $c \xrightarrow{t,\alpha}_1 c'$, and $c' \approx s'$.

Theorem A.1 Given an Orc program *Orc*, let $\mathcal{O}_{Orc} \mathrel{\widehat{=}} (O, o_0, \mathbb{T} \times \Sigma, \longrightarrow_1)$ be the transition system associated with the expression. Let $\mathcal{A}_{Orc}$ be the corresponding Timed Automaton defined using Definition 5 to 11 in the paper. Let $\mathcal{T}_{Orc} \mathrel{\widehat{=}} (S, s_0, \mathbb{T} \times \Sigma, \longrightarrow_2)$ 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 programs. To abuse notations, we write $Orc \approx \mathcal{A}_{Orc}$ to mean $\mathcal{O}_{Orc}.o_0 \approx \mathcal{T}_{Orc}.s_0$.

- $\mathbf{0}$: In Orc semantics, $\mathbf{0}$ has no observable transitions, so $\mathcal{O}_{\mathbf{0}}$ is a single state transition system without any transitions. The same is $\mathcal{T}_{\mathbf{0}}$. Thus, $\mathbf{0} \approx \mathcal{A}_{\mathbf{0}}$.
- $let(z)$: In Orc semantics, the only transition for $\mathcal{O}_{let(z)}$ is $let(z) \xrightarrow{0,z}_1 \mathbf{0}$. It is also the only transition in the responding Timed Automaton. Thus, $let(z) \approx \mathcal{A}_{let(z)}$.

- *Rtimer*(t): In Orc semantics [WKCM08], 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,

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

  The calling site is blocked until the t time units has elapsed. By Definition 6 and 4, the Timed Automaton bi-simulates the site *Rtimer*(t).
- The proof for fundamental sites *if* and *Signal* are skipped for simplicity. The proof is similar to *let*(z) and *Rtimer*(t).
- Site call $M(P)$: According to Orc's operational semantics [WKCM08], the transitions in $\mathcal{O}_{M(P)}$ are $M(P) \xrightarrow{0,\tau}_1 ?k$ and $?k \xrightarrow{t,!m}_1 \mathbf{0}$. According to our Definition 7, the two transitions have one-to-one correspondence to the transitions in the Timed Automaton shown in Fig. 3. In particular, $s_2 \approx \mathbf{0}$ 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:

  $$f >x> g \xrightarrow{t,a}_1 f' >x> g \ \ \textit{if} \ \ f \xrightarrow{t,a}_1 f'$$
  $$f >x> g \xrightarrow{t,\tau}_1 (f' >x> g) \mid [m/x].g \ \ \textit{if} \ \ f \xrightarrow{t,!m}_1 f'$$

  Assume $f \approx \mathcal{A}_f$ and $g \approx \mathcal{A}_g$. For every a such that if $f \xrightarrow{t,a}_1 f'$, there is a transition in $\mathcal{O}_{f>x>g}$. Because $\mathcal{A}_{f>x>g}$ is $\mathcal{A}_f \parallel \mathcal{A}'_g$ (by Definition 8), there is a corresponding transition in $\mathcal{A}_{f>x>g}$ because a is local to automaton $\mathcal{A}_f$ and by Definition 3 the local actions are free to occur. Moreover, $f' \approx \mathcal{A}_{f'}$ by assumption. If $f \xrightarrow{t,!m}_1 f'$, then $f >x> g \xrightarrow{t,!m}_1 (f' >x> g) \mid [m/x].g$. By Definition 8, there is a corresponding transition in $\mathcal{A}'_g$. As long as the number of *get* events are finite, there is always a corresponding transition in one of the $\mathcal{A}'_g$.
  In the other direction, for every transition a from the initial state of $\mathcal{A}_{f>x>g}$, if a is a *get* event, it must be a synchronization between $\mathcal{A}_f$ and one of the $\mathcal{A}'_g$. By assumption, there must be a transition $f \xrightarrow{t,a}_1 f'$. Therefore, there is a corresponding transition in $f >x> g \xrightarrow{t,!m}_1 (f' >x> g) \mid [m/x].g$. 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 *get* event. There must be a corresponding transition in $\mathcal{O}_f$ and $\mathcal{O}_{f>x>g}$. 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:

  $$f \mid g \xrightarrow{t,a}_1 f' \mid g^t \ \ \textit{if} \ \ f \xrightarrow{t,a}_1 f'$$
  $$f \mid g \xrightarrow{t,a}_1 f^t \mid g' \ \ \textit{iff} \ \ g \xrightarrow{t,a}_1 g'$$

  Therefore, f and g are interleaving. By Definition 9, 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. Note that the clocks in the two automata change at same pace so that the timed shift are satisfied automatically. Assume $f \approx \mathcal{A}_f$ and $g \approx \mathcal{A}_g$. By Definition 3 and the above, transitions rules, we conclude $f \mid g \approx \mathcal{A}_{f|g}$.
- Asymmetric composition $f <x< g$: According to the operational semantics of Orc, the two transitions available for the sequential composition are:

  $$f <x< g \xrightarrow{t,a}_1 f' <x< g^t \ \ \textit{if} \ \ f \xrightarrow{t,a}_1 f'$$
  $$f <x< g \xrightarrow{t,\tau}_1 [m/x].f^t \ \ \textit{if} \ \ g \xrightarrow{t,!m}_1 g'$$
  $$f <x< g \xrightarrow{t,a}_1 f^t <x< g' \ \ \textit{if} \ \ g \xrightarrow{t,a}_1 g' \ \textit{and} \ a \neq !m$$

  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 $\mathcal{O}_{f<x<g}$ is the production of $\mathcal{O}_f$ and $\mathcal{O}_g$, where they synchronized on the transition get_x and g is stopped after the synchronization. $\mathcal{T}_{f<x<g}$ is exactly the same transition according to the Definition 9, which uses the shared flag to stop the execution of g. Note that the clocks in the two automata change at same pace so that the timed shift are satisfied automatically.

- Expression call $E(P) \mathrel{\widehat{=}} f$: According to the operational semantics of Orc, the transition available for expression call composition is: $E(P) \xrightarrow{0,\tau}_1 [P/x].f$ *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 Timed Automata model is $call_E(P)$ event in the Definition 11. The one-to-one mapping is shown in the following two transition systems.

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

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

Acknowledgements

The authors would like to thank Prof. Jayadev Misra for insightful discussion on the Orc language and pointing out relevant papers. This work is supported by National Natural Science Fund of China under grants 61100059 and research project "Automatic Checking and Verification of Security Protocol Implementations".

References

[AD94] Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235

[ADW00] Amnell T, David A, Wang Y (2000) A Real-Time Animator for Hybrid Systems. In: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 00), pp 134–145

[AFP+02] Amnell T, Fersman E, Pettersson P, Sun H, Wang Y (2002) Code synthesis for timed automata. Nordic J Comput 9(4):269–300

[AM07] AlTurki M, Meseguer J (2007) Real-time Rewriting Semantics of Orc. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 07), pp 131–142

[AM08] AlTurki M, Meseguer J (2008) Reduction semantics and formal analysis of orc programs. Electr Notes Theor Comput Sci 200(3):25–41

[AM10] AlTurki M, Meseguer J (2010) Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis. Technical report, The University of Illinois at Urbana-Champaign, April https://www.ideals.illinois.edu/handle/2142/15414.

[ASAA08] Ait-Sadoune I, Ait-Ameur Y (2008) Verification and validation of web services composition using the event b method. In Proceedings of the International Summer School about Modeling and Verifying parallel Processes (MOVEP 08), pp 317–322

[ASAA09] Ait-Sadoune I, Ait-Ameur Y (2009) A proof based approach for modelling and verifying web services compositions. In: 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 09), pp 317–322

[BMT06] Bruni R, Melgratti H, Tuosto E (2006) Translating Orc Features into Petri Nets and the Join Calculus. In: Proceeding of the 3rd International Workshop on Web Services and Formal Methods (WS-FM 06), Springer, New York, pp 123–137

[Bro99] Brooke P (1999) A Timed Semantics for a Hierarchical Desgn Notation. PhD thesis, University of York, New York

[BT08a] Borger E, Thalheim B (2008) A method for verifiable and validatable business process modeling. Ad Softw Eng 5316:59–115

[BT08b] Borger E, Thalheim B (2008) Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In: Abstract State Machines (ASM), B and Z (ABZ 08), vol 5238 of LNCS, Springer, New York, pp 24–38

[CCO+04] Chaki S, Clarke EM, Ouaknine J, Sharygina N, Sinha N (2004) State/Event-based Software Model Checking. In: Proceeding of International Conference on Integrated Formal Methods (IFM 04), pp 128–147

[CM05] Cook WR, Misra J (2005) A Structured Orchestration Language. Available for download at http://www.cs.utexas.edu/users/wcook/projects/orc.

[DHQ+04] Song Dong J, Hao P, Qin S, Sun J, Wang Y (2004) Timed Patterns: TCOZ to Timed Automata. In: Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 04), vol 3308 of LNCS, Springer, New York, pp 483–498

[DHQ+08] Dong JS, Hao P, Qin SC, Sun J, Yi W (2008) Timed automata patterns. IEEE Trans Softw Eng 34(6):844–859

[DHSZ06] Dong JS, Hao P, Sun J, Zhang X (2006) A Reasoning Method for Timed CSP Based on Constraint Solving. In: Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 06), vol 4260 of LNCS, Springer, New York, pp 342–359

[DLSZ] Dong JS, Liu Y, Sun J, Zhang X Orc Verification Project Website. http://www.comp.nus.edu.sg/~pat/orc/.

[DLSZ06] Dong JS, Liu Y, Sun J, Zhang X (2006) Verification of computation orchestration via timed automata. In: Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 06), vol 4260 of LNCS, Springer, New York, pp 226–245

[FEK+07] Foster H, Emmerich W, Kramer J, Magee J, Rosenblum DS, Uchitel S (2007) Model Checking Service Compositions under Resource Constraints. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 07), pp 225–234

[FGV04] Farahbod R, Glasser U, Vajihollahi M (2004) Specification and Validation of the Business Process Execution Language for Web Services. In Abstract Sate Machines (ASM 04), vol 3052 of LNCS, Springer, New York, pp 78C94

[Fos08a] Howard Foster M (2008) Tool Support for Safety Analysis of Service Composition and Deployment Models. In: Proceedings of the IEEE International Conference on Web Services (ICWS 08), pp 716–723

[Fos08b] Foster H (2008) WS-Engineer 2008. In: Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC 08), vol 5364 of LNCS, Springer, pp 728–729

[FUMK03] Foster H, Uchitel S, Magee J, Kramer J (2003) Model-based Verification of Web Service Compositions. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 03), pp 152–163

[FUMK06] Foster H, Uchitel S, Magee J, Kramer J (2006) LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: Proceedings of the 28th International Conference on Software Engineering (ICSE 06), pp 771–774

[FV06] Farahbod UGR, Vajihollahi M (2006) An abstract machine architecture for web service based business process management. Int J Bus Process Integr Manag 1(4):279C291

[GG09] Göthel T, Glesner S (2009) Machine Checkable Timed CSP. In Proceedings of the 1st NASA Formal Methods Symposium (NFM 09). NASA Conference Publication, NASA

[GP97] Gupta Gl, Pontelli E (1997) A Constraint-based Approach for Specification and Verification of Real-time Systems. In: IEEE Real-Time Systems Symposium, pp 230–239

[HNSY92] Henzinger TA, Nicollin X, Sifakis J, Yovine S (1992) Symbolic Model Checking for Real-Time Systems. In: Proceedings of the 7th International Symposium of Logics in Computer Science (LICS 92), pp 394–406

[Hoa85] Hoare CAR (1985) Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall

[JJV05] Santosa A, Jaffar J, Voicu R (2005) Modeling Systems in CLP with Coinductive Tabling. In: Proceedings of the 21st International Conference on Logic Programming (ICLP 2005), pp 412–413

[JM94] Jaffar J, Maher MJ (1994) Constraint logic programming: a survey. J Log Progra 19/20:503–581

[JMSY92] Jaffar J, Michaylov S, Stuckey PJ, Yap RHC (1992) The CLP(R) Language and System. ACM Trans Program Lang Syst 14(3):339–395

[JSV04] Jaffar J, Santosa AE, Voicu R (2004) A CLP Proof Method for Timed Automata. In: Real-Time Systems Symposium, pp 175–186

[KCM06] Kitchin D, Cook WR, Misra J (2006) A language for task orchestration and its semantic properties. In: Proceedings of the International Conference on Concurrency Theory (CONCUR 06), pp 477–491

[LPW97] Larsen KG, Pettersson P, Wang Y (1997) Uppaal in a Nutshell. Intern J Softw Tool Technol Trans 1(1-2):134–152

[LPY95] Larsen KG, Pettersson P, Yi W (1995) Model-Checking for Real-Time Systems. In: Proceedings of Fundamentals of Computation Theory, number 965 in LNCS, pp 62–88

[LSD10] Liu Y, Sun J, Dong JS (2010) Developing model checkers using pat. In: Proceedings of the 8th International Symposium of Automated Technology for Verification and Analysis (ATVA 10), Springer, pp 371–377

[LSD11] Liu Y, Sun J, Dong JS (2011) Pat 3: An extensible architecture for building multi-domain model checkers. In: Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE 11), pp 190–199

[LW00] Lin HM, Wang Y (2000) A Proof System for Timed Automata. In: Tiuryn J (ed) Proceedings of the 3rd International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 00), vol 1784 of LNCS, pp 208–222

[LZH10] Li Q, Zhu H, He J (2010) A Denotational Semantical Model for Orc Language. In: Proceedings of the 7th International colloquium conference on Theoretical aspects of computing, ICTAC'10, Springer-Verlag, Heidelberg, pp 106–120

[MC07] Misra J, Cook W (2007) Computation orchestration: a basis for wide-area computing. Softw Syst Model 6(1):83–110

[MHM04] Misra J, Hoare T, Menzel G (2004) A Tree Semantics of an Orchestration Language. In: Proceedings of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, NATO ASI Series, Marktoberdorf, Germany

[Mil89] Milner R (1989) Communication and Concurrency. Prentice-Hall International, Prentice-Hall

[Mil99] Milner R (1999) Communicating and Mobile Systems: the π Calculus. Cambridge University Press, Cambridge

[Nak05] Nakajima S (2005) Model-Checking Behavioral Specification of BPEL Applications. In: Proceeding of the 2nd International Workshop on Web Services and Formal Methods (WS-FM 05), France

[OAS07] OASIS (2007) Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[OW02] Ouaknine J, Worrell J (2002) Timed CSP = closed timed safety automata. Electr Note Theor Comput Sci 68(2):142–159

[PZWQ06] Pu G, Zhao X, Wang S, Qiu Z (2006) Towards the Semantics and Verification of BPEL4WS. Electr Note Theor Comput Sci 151(2):33–52

[Ros97] Roscoe AW (1997) The Theory and Practice of Concurrency. Prentice-Hall

[Sch00] Schmidt K (2000) LoLA: A Low Level Analyser. In: Proceeding of the 21st International Conference of Application and Theory of Petri Nets (ICATPN 00), pp 465–474

[SD95] Schneider S, Davies J (1995) A Brief History of Timed CSP. Theoretical Computer Science 138, Oxford

[SH05] Singh MP, Huhns MN (2005) Service-Oriented Computing. Wiley, Chichester

[SLD$^+$13] Sun J, Liu Y, Dong JS, Liu Y, Shi L, André É (2013) Modeling and verifying hierarchical real-time systems using stateful timed csp. ACM Trans Softw Eng Methodol (TOSEM) 22(1):1–3

[SLDC09] Sun J, Liu Y, Dong JS, Chen C (2009) Integrating specification and programs for system modeling and verification. In: Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 09). IEEE Computer Society, pp 127–135

[SLDP09] Sun J, Liu Y, Dong JS, Pang J (2009) PAT: Towards Flexible Verification under Fairness. In: Proceedings of the 21th International Conference on Computer Aided Verification (CAV 09) volume 5643 of Lecture Notes in Computer Science, pp 709–714

[SMS05] Schlingloff BH, Martens A, Schmidt K (2005) Modeling and model checking web services. In: Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems, pp 27

[Tiu05] Tiu A (2005) Model Checking for Pi-calculus Using Proof Search. In: Proceedings of the International Conference on Concurrency Theory (CONCUR 05), San Francisco

[WCG$^+$06] Wirsing M, Clark A, Gilmore S, Hölzl M, Knapp A, Koch N, Schroeder A (2006) Semantic-Based Development of Service-Oriented Systems. In: Proceeding. 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems (FORTE 06), LNCS 4229, Springer-Verlag, New York, pp 24–45

[WDG$^+$07] Wirsing M, Nicola RD, Gilmore S, Hölzl M, Lucchi R, Tribastone M, Zavattaro G (2007) SENSORIA Process Calculi for Service-Oriented Computing. In: Trustworthy Global Computing, Second Symposium (TGC 06), volume 4661 of LNCS, Springer, pp 30–50

[WHA$^+$08] Wirsing M, Hölzl M, Acciai L, Clark A, Banti F, Fantechi A, Gilmore S, Gnesi S, Gönczy L, Koch N, Lapadula A, Mayer P, Mazzanti F, Pugliese R, Schroeder A, Tiezzi F, Tribastone M, Varró D (2008) A Pattern-Based Approach to Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity. In: Proceedings of 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 08), LNCS. Springer-Verlag, New York

[WKCM08] Wehrman I, Kitchin D, Cook Wr, Misra J (2008) A Timed Semantics of Orc. Theor Comput Sci 402(2–3):234–248

Received 8 August 2011
Revised 20 November 2012
Accepted 5 March 2013 by J. Woodcock
Published online 30 April 2013