10-2011

Verification of orchestration systems using compositional partial order reduction

Tian Huat TAN

Yang LIU

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

Jin Song DONG

Citation

TAN, Tian Huat; LIU, Yang; SUN, Jun; and DONG, Jin Song. Verification of orchestration systems using compositional partial order reduction. (2011). *Proceedings of the 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28*. 98-114.
Available at: https://ink.library.smu.edu.sg/sis_research/5028

Verification of Orchestration Systems Using Compositional Partial Order Reduction*

Tian Huat Tan[1], Yang Liu[2], Jun Sun[3], and Jin Song Dong[2]

[1] NUS Graduate School for Integrative Sciences and Engineering
tianhuat@comp.nus.edu.sg
[2] School of Computing, National University of Singapore
{liuyang,dongjs}@comp.nus.edu.sg
[3] Singapore University of Technology and Design
sunjun@sutd.edu.sg

Abstract. *Orc* is a computation orchestration language which is designed to specify computational services, such as distributed communication and data manipulation, in a concise and elegant way. Four concurrency primitives allow programmers to orchestrate site calls to achieve a goal, while managing timeouts, priorities, and failures. To guarantee the correctness of *Orc* model, effective verification support is desirable. *Orc* has a highly concurrent semantics which introduces the problem of state-explosion to search-based verification methods like model checking. In this paper, we present a new method, called Compositional Partial Order Reduction (CPOR), which aims to provide greater state-space reduction than classic partial order reduction methods in the context of hierarchical concurrent processes. Evaluation shows that CPOR is more effective in reducing the state space than classic partial order reduction methods.

1 Introduction

The advent of multi-core and multi-CPU systems has resulted in the widespread use of concurrent systems. It is not a simple task for programmers to utilize concurrency, as programmers are often burdened with handling threads and locks explicitly. Processes can be composed at different levels of granularity, from simple processes to complete workflows. The *Orc* calculus [17] is designed to specify orchestrations and wide-area computations in a concise and structured manner. It has four concurrency combinators, which can be used to manage timeouts, priorities, and failures effectively [17]. The standard operational semantics [29] of *Orc* supports highly concurrent executions of *Orc* sub-expressions. Concurrency errors are difficult to discover by testing. Hence, it is desirable to verify *Orc* formally. The highly concurrent semantics of *Orc* can lead to state space explosion and thus pose a challenge to model checking methods.

In the literature, various state reduction techniques have been proposed to tackle the state space explosion problem, including on-the-fly verification [15], symmetry reduction [7,11], partial order reduction (POR) [8,22,12,28,5,23], etc. POR works by exploiting the independency of concurrently executing transitions in order to reduce the

* This research is supported in part by Research Grant IDD11100102 of Singapore University of Technology and Design, IDC and MOE2009-T2-1-072 (Advanced Model Checking Systems).

S. Qin and Z. Qiu (Eds.): ICFEM 2011, LNCS 6991, pp. 98–114, 2011.

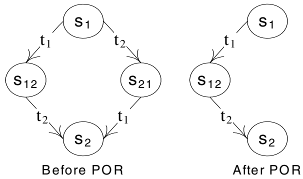

Fig. 1. Partial Order Reduction

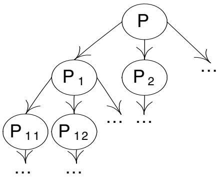

Fig. 2. Hierarchical Concurrent Processes

number of possible interleavings. For example, consider the transition system in Figure 1 where t_1 and t_2 are independent transitions. This means that executing either t_1t_2 or t_2t_1 from state s_1 will always lead to state s_2. POR will detect such independency, and choose only t_1t_2 for execution, thus reducing the explored state space. Classic POR algorithms, such as [28,12,8,22,5], work by identifying a subset of outgoing transitions of a state which are sufficient for verification. In this paper, we denote such subsets as *ample sets* – see [8,5].

Many concurrent systems are designed using a top-down architecture, and concurrent processes are structured in a hierarchical way. In Figure 2, process P contains subprocesses P_i (i = 1, 2, etc.) that are running concurrently. Moreover, each process P_i also contains subprocesses P_{ij} (j = 1, 2, etc.) that are running concurrently. We refer to concurrent processes of this kind as hierarchical concurrent processes (HCP). There are many real-life examples of HCP. Consider a browser that supports tabbed browsing. Multiple browser windows could be opened at the same time, each browser window could contain multiple opened tabs, and each opened tab could download several HTML elements in parallel. *Orc* processes provide another example of HCP.

Classic POR algorithms, such as [28,12,8,22,5], assume that *local transitions* within the participated processes are dependent. In the context of HCP (Figure 2), if POR is applied on process P, transitions within processes P_1, P_2, etc. will be considered as *local transitions*, and be assumed to be dependent. Nevertheless, many local transitions may be independent. In this work, we propose a method called Compositional Partial Order Reduction (CPOR), which extends POR to the context of HCP. CPOR exploits the independency within local transitions. It applies POR recursively for the hierarchical concurrent processes, and several possible ample sets are composed in a bottom-up manner. In order to apply CPOR to *Orc*, we first define the HCP structure of an *Orc* process. Subsequently, based on the HCP structure, we established some local criteria that could be easily checked by CPOR algorithm. Experimental results show that CPOR can greatly reduce the explored state space when verifying Orc models.

Paper Outline. Section 2 introduces *Orc* language. Section 3 elaborates on CPOR and shows how it can be applied to *Orc* models. Section 4 gives several experimental results. Section 5 surveys the related work. Finally, Section 6 discusses the extensibility of CPOR with possible future work and concludes the paper.

2 Orchestration Language Orc

2.1 Syntax

Orc is a computation orchestration language in which multiple services are invoked to achieve a goal while managing time-outs, priorities, and failures of services or communication. Following is the syntax of *Orc*:

$$\begin{array}{llcll}
\textit{Variable} & x & ::= & \textit{variable name} & \\
\textit{Value} & m & ::= & \textit{value} & \\
\textit{Parameter} & p & ::= & x \mid m & \\
\textit{Expression} & E & ::= & M(\overline{p}) & \text{– site call} \\
 & & \mid & E \mid E & \text{– parallel} \\
 & & \mid & E > x > E & \text{– sequential} \\
 & & \mid & E < x < E & \text{– pruning} \\
 & & \mid & E \;;\; E & \text{– otherwise}
\end{array}$$

Site. The simplest *Orc* expression is a site call $M(\overline{p})$, where M is the service's name and $\overline{p}$ is a list of parameters. Sites are the basic units of *Orc* language. A site can be an external service (e.g. *Google* site) which resides on a different machine. For example, *Google*(*"Orc"*) is an external site call that calls the external service provided by *Google* and its response is the search results for keyword *"Orc"* by the *Google* search engine. A site can also be a local service (e.g. *plus* site) which resides on the same machine. For example, a site call *plus*$(1, 1)$ calls the local *plus* service and its response is the summation of the two arguments. Since a site in *Orc* is essentially a service, henceforth, we would use the term *site* and *service* interchangeably. Some services maintain a state, those services are denoted as *stateful services*. An example is *Buffer* site, which provides the service of First-In-First-Out (FIFO) queue. We denote the data structure that constitutes the state of a stateful service as *state object* of the stateful service. A site call (e.g. a dequeue operation on *Buffer* site) for a certain stateful service may change the corresponding state object (e.g. a FIFO queue). Thus, multiple site calls with the same arguments to the same stateful service might result in different responses. Services that do not have any state are called *stateless services*. An example is *plus* site, which takes two numbers as input and returns their summation. Multiple calls with the same arguments to a stateless service will always result in the same response.

Combinators. There are four combinators: parallel, sequential, pruning, and otherwise combinators. The parallel combinator $F \mid G$ defines a parallel expression, where expressions F and G execute independently, and its published value can be the value published either by F or by G or both of them. The sequential combinator $F > x > G$ defines a sequential expression, where each value published by F initiates a separate execution of G wherein x is bound to the published value. The execution of F is then continued in parallel with all these executions of G. The values published by the sequential expression are the values published by the executions of G. For example, $(\textit{Google}(\text{“}\textit{Orc}\text{”}) \mid \textit{Yahoo}(\text{“}\textit{Orc}\text{”})) > x > \textit{Email}(\textit{addr}, x)$ will call Google and Yahoo sites simultaneously. For each returned value, an instance of x will be bound to it, and an email will be sent to *addr* for each instance of x. Thus, up to two emails will be sent. If x is not used in G, $F \gg G$ can be used as a shorthand for $F > x > G$.

The pruning combinator $F < x < G$ defines a pruning expression, where initially F and G execute in parallel. However, when F needs the value of x, it will be blocked until G publishes a value to bind x and G terminates immediately after that. For example, $Email(addr, x) < x < (Google(``Orc") \mid Yahoo(``Orc"))$ will get the fastest searching result for the email sending to *addr*. If x is not used in F, $F \ll G$ can be used as a shorthand for $F < x < G$. The otherwise combinator $F \ ; \ G$ defines an otherwise expression, where F executes first. The execution of F is replaced by G if F halts without any published value, otherwise G is ignored. For example, in the expression $(Google(``Orc") \ ; \ Yahoo(``Orc")) > x > Email(addr, x)$, Yahoo site is used as a backup service for searching "*Orc*" and it will be called only if the site call $Google(``Orc")$ halts without any result for "*Orc*".

Functional Core Language (Cor). *Orc* is enhanced with functional core language (Cor) to support various data types, mathematical operators, conditional expressions, function calls, etc. Cor structures such as conditional expressions and functions are translated into site calls and four combinators [17]. For example, conditional expression *if E then F else G*, where E, F, and G are *Orc* expressions would be translated into expression $(if(b) \gg F \mid if(\sim b) \gg G) < b < E$ before evaluation.

Example - Metronome. Timer is explicitly supported in *Orc* by introducing time-related sites that delay a given amount of time. One of such sites is *Rtimer*. For example, $Rtimer\ (5000) \gg ``Orc"$ will publish "Orc" at exactly 5 seconds. Functional core (Cor) defines functions using the keyword *def*. Following is a function that defines a metronome [17], which will publish a *signal* value every *t* seconds. *signal* is a value in *Orc* that carries no information. Note that the function is defined recursively.

$$def\ metronome(t) \ = \ (signal \ \mid \ Rtimer(t) \gg metronome(t))$$

The following example publishes "tick" once per second, and publishes "tock" once per second after an initial half-second delay.

$$(metronome(1000) \gg ``tick") \mid (Rtimer(500) \gg metronome(1000) \gg ``tock")$$

Thus the publications are "tick tock tick $\cdots$" where "tick" and "tock" alternate each other. One of the properties that we are interested is whether the system could publish two consecutive "tick"s or two consecutive "tock"s which is an undesirable situation. In order to easily assert a global property that holds throughout the execution of an *Orc* program, we extend *Orc* with auxiliary variables. The value of an auxiliary variable could be accessed and updated throughout the *Orc* program. Henceforth, we will simply refer to the extended auxiliary variables as *global variables*. A global variable is declared with the keyword *globalvar* and a special site, *\$GUpdate*, is used to update a global variable. We augment the metronome example with a global variable *tickNum*, which is initialized to zero. *tickNum* is increased by one when a "tick" is published, and is decreased by one when a "tock" is published.

$globalvar\ tickNum = 0$
$def\ metronome(t) = (signal \mid Rtimer(t) \gg metronome(t))$
$(metronome(1000) \ \gg \ \$GUpdate(\{tickNum = tickNum + 1\}) \gg \ ``tick")$
$\mid \ (Rtimer(500) \ \gg \ metronome(1000) \ \gg \ \$GUpdate(\{tickNum = tickNum - 1\})$
$\gg \ ``tock")$

With this, we are allowed to verify whether the system could publish two consecutive "tick"s or two consecutive "tock"s by checking the temporal property such that whether the system is able to reach an undesirable state that satisfying the condition $(\mathit{tickNum} < 0 \vee \mathit{tickNum} > 1)$.

2.2 Semantics

This section presents the semantic model of *Orc* based on Label Transition System (LTS). In the following, we introduce some definitions required in the semantic model.

Definition 1 (System Configuration). *A system configuration contains two components (Proc,Val), where Proc is a Orc expression, and Val is a (partial) variable valuation function, which maps the variables to their values.*

A variable in the system could be an *Orc*'s variable, or the global variable which is introduced for capturing global properties. The value of a variable could be a primitive value, a reference to a site, or a state object. The three primitive types supported by *Orc* are boolean, integer, and string. All variables are assumed to have finite domain. Two configurations are equivalent iff they have the same process expression *Proc* and same valuation function *Val*. *Proc* component of system configuration is assumed to have finitely many values.

Definition 2 (System Model). *A system model is a 3-tuple $\mathcal{S} = (Var, init_G, P)$, where Var is a finite set of global variables, $init_G$ is the initial (partial) variable valuation function and P is the Orc expression.*

Definition 3 (System Action). *A system action contains four components (Event, Time, EnableSiteType, EnableSiteId). Event is either publication event, written $!m$ or internal event, written τ. EnableSiteType, EnableSiteId are the type and unique identity of the site that initiates the system action. Time is the total delay time in system configuration before the system action is triggered.*

Every system action is initiated by a site call, and we extend the system action defined in [29] with two additional components, *EnableSiteType* and *EnableSiteId*, to provide information for CPOR. A publication event $!m$ communicates with the environment with value *m*, while an internal event τ is invisible to the environment. There are three groups of site calls. The first two groups are site calls for stateless and stateful services respectively. And the third are the site calls for $\$GUpdate$ which update global variables. These three groups are denoted as *stateless*, *stateful*, and *GUpdate* respectively, and those are the possible values for *EnableSiteType*. Every site in the system model is assigned a unique identity which ranges over non-negative integer value. Discrete time semantics [29] is assumed in the system. *Time* ranges over non-negative integer value and is assumed to have finite domains.

Definition 4 (Labeled Transition System (LTS)). *Given a model $\mathcal{S} = (Var, init_G, P)$, let Σ denote the set of system actions in P. The LTS corresponding to $\mathcal{S}$ is a 3-tuple $(C, init, \rightarrow)$, where C is the set of all configurations, $init \in C$ is the initial system configuration $(P, init_G)$, and $\rightarrow \subseteq C \times \Sigma \times C$ is a labeled transition relation, and its definition is according to the operational semantics of Orc [29].*

To improve readability, we write $c \xrightarrow{a} c'$ for $(c, a, c') \in \rightarrow$. An action $a \in \Sigma$ is *enabled* in a configuration $c \in C$, denoted as $c \xrightarrow{a}$, iff there exists a configuration $c' \in C$, such that $c \xrightarrow{a} c'$. An action $a \in \Sigma$ is *disabled* in a configuration $c = (P, V)$, where $c \in C$, iff the action a is not enabled in the configuration c, but it is enabled in some configurations (P, V'), where $V' \neq V$. $Act(c)$ is used to denote the set of enabled actions of a configuration $c \in C$, formally, for any $c \in C$, $Act(c) = \{a \in \Sigma \mid c \xrightarrow{a}\}$. $Enable(c, a)$ is used to denote the set of reachable configurations through an action $a \in \Sigma$ from a configuration $c \in C$, that is, for any $c \in C$ and $a \in \Sigma$, $Enable(c, a) = \{c' \in C \mid c \xrightarrow{a} c'\}$. $Enable(c)$ is used to denote the set of reachable configurations from a configuration $c \in C$, that is, for any $c \in C$, $Enable(c) = \{c' \in Enable(c, a) \mid a \in \Sigma\}$. $Ample(c)$ is used to denote the ample set (refer to Section 3) of a configuration $c \in C$. $AmpleAct(c)$ is defined as the set of actions that caused a configuration $c \in C$ transit into the configurations in $Ample(c)$, that is, for any $c \in C$, $AmpleAct(c) = \{a \in \Sigma \mid c \xrightarrow{a} c', c' \in Ample(c)\}$. $PAct(c)$ is used to denote the set of enabled and disabled actions of a configuration c, and $Act(c) \subseteq PAct(c)$. We use *TS* to represent the original LTS before POR is applied and $\widehat{TS}$ to represent the reduced LTS after POR is applied. TS_c is used to represent the LTS (before any reduction) that starts from c, where c is a configuration in *TS*. An *execution fragment* $l = c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \ldots$ of LTS is an alternating sequence of configurations and actions. A *finite execution fragment* is an execution fragment ending with a configuration.

We are interested in checking the system against two kinds of properties. The first kind is deadlock-freeness, which is to check whether there does not exist a configuration $c \in C$ in *TS* such that $Enable(c) = \varnothing$. The second kind is temporal properties that are expressible with LTL without Next Operator (LTL-X) [5]. For any LTL-X formula ϕ, prop(ϕ) denotes the set of atomic propositions used in ϕ. In the metronome example which augmented with a global variable *tickNum*, prop(ϕ)=$\{(tickNum < 0), (tickNum > 1)\}$. An action $a \in \Sigma$ is *ϕ-invisible* iff the action does not change the values of propositions in prop(ϕ) for all $c \in C$ in *TS*.

2.3 Hierarchical Concurrent Processes (HCP)

The general structure of a hierarchical concurrent process P is shown graphically using a tree structure in Figure 3. Henceforth, we denote such a graph as a HCP graph, or simply HCP if it does not lead to ambiguity.

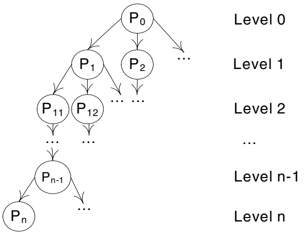

Fig. 3. The general structure of HCP

Figure 3 shows that process P_0 contains subprocesses P_1, P_2, etc that are running concurrently. Process P_1 in turn contains subprocesses P_{11}, P_{12}, etc that are running concurrently. This goes repeatedly until reaching a process P_n which has no subprocesses. Each process P in the hierarchy will have its associated level, starting from level 0. A process without any subprocess (e.g. process P_n) is denoted as *terminal process*, otherwise the process is denoted as *non-terminal process*. Furthermore, process P_0 at level 0 is denoted as *global process*, while processes at level i, where $i > 0$, are denoted as *local processes*. The *parent process* of a local process P' is a unique process P such that there is a directed edge from P to P' in the HCP graph. When P is the parent process of P', P' is called the *child process* of P. *Ancestor processes* of a local process P' are the processes in the path from global process to P'. *Descendant processes* of process P are those local processes that have P as an *ancestor process*.

An *Orc* expression P could be viewed as a process that is composed by HCP. This could be formalized by constructing the HCP according to syntax of P, assigning process identity to each sub-expression of P, and defining how the defined processes evolve during the execution of expression P. In the following, we illustrate this in detail. An *Orc* expression can be either a site call or one of the four combinators and their corresponding HCPs are shown in Figure 4. A site call is a terminal process node, while each of the combinators has either one or two child processes according to their semantics (refer to Section 2), and the HCPs of respective child process nodes are defined recursively. We denote expressions A and B as LHS process and RHS process for each combinators in Figure 4. For example, a pruning combinator ($A < x < B$) contains two child nodes because its LHS process and RHS process could be executed concurrently. Each of the process nodes in HCP is identified by a unique process identity (*pid*), and node values in HCP are prefixed with their *pid* (e.g. p_0, p_1, etc.). In Figure 5, an expression $(S_1 \ll S_2) \mid (S_3 \ll S_4)$, where S_1, S_2, S_3, and S_4 are site calls, could be viewed as a process composed by HCP of three levels.

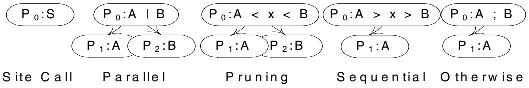

Fig. 4. HCP of general Orc Expressions

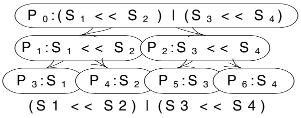

Fig. 5. An example

Consider a transition $(P, V) \xrightarrow{a} (P', V')$, where a is some action. We abuse the notation by using P and P' to denote the HCPs before and after the transition. In fact, P' could have different tree structures from P, and processes could be added or deleted in P'. In order to have a clear relation of processes between P and P', we define the relation of processes between P and P' over each rule of the operational semantics of *Orc* [27], some of which are presented in Figure 6 for illustration purpose. There are two HCPs under each rule. HCPs on the left and right are the HCPs before and after triggering the action initiated by respective rules. Two process nodes on different HCPs belong to the same process if they have the same *pid* value, and an arrow is used to relate them. Processes that could only be found in HCP on the right or left are the processes that are newly added or deleted respectively. In SEQ1V, the transition of f to f' produces an

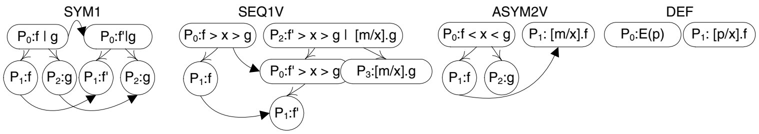

Fig. 6. Relation of Processes between P and P'

$$A = (\mathit{userdb.put}(\text{“}\mathit{user}1\text{”}) \mid \mathit{userdb.put}(\text{“}\mathit{user}2\text{”})) < \mathit{userdb} < \mathit{Buffer}()$$
$$B = (\mathit{flightdb.put}(\text{“}\mathit{CX}510\text{”}) \mid \mathit{flightdb.put}(\text{“}\mathit{CX}511\text{”})) < \mathit{flightdb} < \mathit{Buffer}()$$

Fig. 7. Execution of *Orc* process $P = A \mid B$

output value m, and notation $[m/x].g$ is used to denote that all the instances of variable x in g are replaced with value m.

A site S is *private in* $P_1[P]$, if the reference of site S could not be accessed by all processes other than process P_1 and its descendant processes under HCP graph of global process P. Otherwise, site S is *shared in process* $P_1[P]$. A site S is *permanently private in* $P_1[c]$, if for any configuration $c' = (P', V')$ that is reachable by c, if P' has P_1 as its descendant process, site S must be private in process $P_1[P']$.

The example in Figure 7 shows an *Orc* process $P = A \mid B$. Variables *userdb* and *flightdb* will be initialized to different instances of site *Buffer*, which provides the service of FIFO queue. In process A, two string values *user*1 and *user*2 are enqueued in the buffer referenced by *userdb* concurrently. *Buffer* site that is referenced by *userdb* is *private* in $A[P]$, since *userdb* could only be accessed by process A. Now consider at some level j of HCP graph of global process P, where $j > 1$, we have processes $P_{j_1} = \mathit{userdb.put}(\text{“}\mathit{user}1\text{”})$ and $P_{j_2} = \mathit{userdb.put}(\text{“}\mathit{user}2\text{”})$. *Buffer* site that is referenced by *userdb* is *shared* in $P_{j_1}[P]$, since *userdb* could be accessed by P_{j_2} which is not a descendant process of P_{j_1}.

3 Compositional Partial Order Reduction (CPOR)

The aim of Partial Order Reduction (POR) is to reduce the number of possible orderings of transitions by fixing the order of independent transitions as shown in Figure 1. The notion of *indepedency* plays a central role in POR, which is defined below by following [13].

Definition 5 (Independency). *Two actions a_1 and a_2 in an LTS are independent if for any configuration c such that a_1, $a_2 \in Act(c)$:*
1. $a_2 \in Act(c_1)$ where $c_1 \in Enable(c, a_1)$ and $a_1 \in Act(c_2)$ where $c_2 \in Enable(c, a_2)$,
2. Starting from c, any configuration reachable by executing a_1 followed by a_2, can also be reached by executing a_2 followed by a_1.
Two actions are dependent iff they are not independent.

Given a configuration, an ample set is a subset of outgoing transitions of the configuration which are sufficient for verification, and it is formally defined as follow:

Definition 6 (Ample Set). *Given an LTL-X property ϕ, and a configuration $c \in C$ in TS, an* ample set *is a subset of the enable set which must satisfy the following conditions [5]:*
(A1) Nonemptiness condition: *$Ample(c) = \varnothing$ iff $Enable(c) = \varnothing$.*
(A2) Dependency condition: *Let $c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} c_n \xrightarrow{a} t$ be a finite execution fragment in TS. If a depends on some actions in $AmpleAct(c_0)$, then $a_i \in AmpleAct(c_0)$ for some $0 < i \leq n$.*
(A3) Stutter condition: *If $Ample(c) \neq Enable(c)$, then any $\alpha \in AmpleAct(c)$ is ϕ-invisible.*
(A4) Strong Cycle condition: *Any cycle in $\widehat{TS}$ contains at least one configuration c with $Ample(c)$=$Enable(c)$.*

To be specific, reduced LTS generated by the ample set approach needs to satisfy conditions A1 to A4 in order to preserve the checking of LTL-X properties. However, for the checking of deadlock-freeness, only conditions A1 and A2 are needed [12]. Henceforth, our discussion will be focused on the checking of LTL-X property, but the reader could adjust accordingly for the checking of deadlock-freeness.

Conditions A1, A3, and A4 are relatively easy to check, while condition A2 is the most challenging condition. It is known that checking condition A2 is equivalent to checking the reachablity of a condition in the full transition system TS [8]. It is desirable that we could have an alternative condition A2' that only imposes requirements on the current configuration instead of all traces in TS, and satisfaction of condition A2' would guarantee the satisfaction of condition A2. Given a configuration $c_g = (P_g, V_g)$, and P_d as a descendant process of P_g, with associated configuration $c_d = (P_d, V_d)$, we define a condition A2' that based solely on c_d, and its soundness will be proved in Section 3.3.

(A2')Local Criteria of A2. For all configurations $c_a \in Ample(c_d)$ and $c_a = (p_a, v_a)$ the following two conditions must be satisfied:
(1) The enable site for the action a that enable c_a must be either stateless site, or stateful site *private in* $p_a[P_g]$;
(2) p_a is not a descendant process of the RHS process of some pruning combinators or the LHS process of some sequential combinators.

Notice that we define an ample set as a set of enabled configurations rather than a set of enabled actions like [5]. The reason is due to in references like [5], action-deterministic system is assumed. This entails that for any configuration $c \in C$ and any action $a \in \Sigma$, c has at most one outgoing transition with action a, formally, $c \xrightarrow{a} c'$ and $c \xrightarrow{a} c''$ implies $c' = c''$. Therefore, the enabled configurations could be deduced by the enabled actions. Nonetheless, an *Orc* system is not action-deterministic, the main reason is because some events in *Orc* are internal events that are invisible to the environment. By defining ample set as a set of configurations, with their associated enabled actions, the requirement of action-deterministic system is no longer needed.

3.1 Classic POR and CPOR

Classic POR methods assume that local transitions of a process are dependent, and in the context of HCP, it means that actions within individual processes from level 1

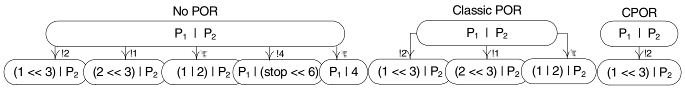

Fig. 8. LTS of *Orc* Process $P = (P_1 \mid P_2), P_1 = ((1 \mid 2) \ll 3), P_2 = (4 \ll 6)$

onwards are *simply assumed to be dependent*. In Figure 8, three LTSs of the process P are given. No POR shows the set of all initial transitions of process P; classic POR shows how the state-space of a parallel composition can be reduced when its component processes are independent; and CPOR reduces the initial actions further by examining internal process structure. For simplicity, system configuration is represented only by process expression. When no POR is applied, all interleavings of transitions are considered, and there are five branches after the initial state. When the classic POR is applied, since P_1 and P_2 are active processes, assume that it checks process P_1 first. All transitions of P_1 are assumed to be dependent by the classic POR. For this reason the resulting ample set of P is $\{((1 \ll 3) \mid P_2), ((2 \ll 3) \mid P_2), ((1 \mid 2) \mid P_2)\}$, which is a valid ample set after checking for conditions A1-A4. Therefore, there are three branches from initial state when classic POR is applied. Different from classic POR, when CPOR is applied, POR is again applied to process $(1 \mid 2)$. We define $Amples(P)$ as a set of ample sets of process P that satisfy conditions A1 and A2, but yet to be checked for conditions A3 and A4. $Amples((1 \mid 2)) = \{\{1\}, \{2\}\}$ and $Amples(P_1)$ is $Amples((1 \mid 2))$ after restructuring by the semantics of P_1, which is $\{\{1 \ll 3\}, \{2 \ll 3\}\}$. $Amples(P)$ is $Amples(P_1)$ after restructuring by the semantics of P, which is $\{\{1 \ll 3 \mid P_2\}, \{2 \ll 3 \mid P_2\}\}$. Each ample set in $Amples(P)$ will then be checked for conditions A3 and A4, and both ample sets turn up to be valid, therefore the ample set $\{1 \ll 3 \mid P_2\}$ is chosen nondeterministically to be the returned value. Thus there is only a single branch after the initial state when CPOR is applied. There are a total of 31, 14 and 5 states for LTS of process P in the situations where no POR, classic POR and CPOR are applied respectively.

3.2 CPOR Algorithm

In this section, we discuss the procedures for CPOR as given in Algorithm 1. *CAmple* returns an ample set which is a subset of enabled configurations from the configuration $c = (P, V)$, and *Visited* is the stack of previously visited configurations. Each configuration c_a in the ample set, where $c_a = (Proc, Val)$, is associated with an action $a_a = (Event, Time, EnableSiteType, EnableSiteId)$, which caused the transition from c to c_a, that is $c \xrightarrow{a_a} c_a$. Henceforth, we use the dot-notation such as $c_a.Proc$, $c_a.Event$, etc to denote the component values of c_a as well as the component values of its associated action a_a. *P.Amples* (line 2) is a set that stores ample set candidates that satisfy conditions A1 and A2, but yet to check for conditions A3 and A4. Procedure *enableSubProcs*(P) (line 3) returns the set of enabled child processes according to HCP graph of *Orc* expressions P as shown in Figure 4, with an exception that for sequential process $P_s = A > x > B$, it returns an empty set $\{\}$ instead of $\{A\}$, and for pruning process $P_p = A < x < B$, it returns $\{A\}$ instead of $\{A, B\}$. This exception is applied in order to satisfy the condition A2'(2). Procedure $\mathit{fillAmpleRec}(P,V)$ (line 17) retrieves the ample set candidates under valuation V and assigns it to $P.\mathit{Amples}$. In line 18, $\mathit{Enable}(c)$ where $c = (P,V)$ gives the set of all enabled configurations from the configuration c. Procedure $\mathit{checkA2Local}(\mathit{config})$ checks whether configuration *config* satisfies A2'(1). Procedure *isPrivate* (line 32) checks whether the site with *config.EnableSiteId* as unique identity is *private* in $\mathit{Proc}[P_G]$ where *Proc* is the process component of *config* and P_G is the argument P of procedure *CAmple* provided by user, which is the global process that has *Proc* as descendant process. The checking is done by syntax analysis. In *Orc*, P is a terminal process (line 20) iff it is a site call. Procedure $\mathit{composeAmples}(P,sP,V)$ (line 26) combines $sP.\mathit{Amples}$ back into $P.\mathit{Amples}$ under valuation V. Procedure *reformAmples* $(sP.\mathit{Amples},P)$ (line 27) restructures configurations within *sp.Amples* by operational semantics of *Orc*. For example, consider $P = (1 + x < x < 2)$, and $sP = 2$. After making a transition, $sP.\mathit{Amples} = \{\{c\}\}$, where c is the configuration $(\mathit{stop},\varnothing)$ with $c.\mathit{Event} = !2$. After restructuring by $\mathit{reformAmples}(sP.\mathit{Amples},P)$, c becomes $(1+2,\varnothing)$, and $c.\mathit{Event} = \tau$, according to rule $\mathit{ASYM2V}$ as stated below.

$$\frac{(2,\varnothing) \xrightarrow{!2} (\mathit{stop},\varnothing)}{(1+x<x<2,\varnothing) \xrightarrow{\tau} (1+2,\varnothing)} \quad [\,\mathit{ASYM2V}\,]$$

When $P = sP$, $\mathit{reformAmples}(sP.\mathit{Amples},P)$ will simply return $sP.\mathit{Amples}$. Subsequently, ample sets that are empty sets are filtered away (line 28). We continue on the discussion of procedure *CAmple*. To analyze whether an ample set *ample* is valid, the algorithm checks whether all configurations within satisfy conditions A3 and A4 (line 9, 10). If it turns out to be true, a valid ample set is found, and it will be returned immediately (line 14, 15). If no valid ample set has been found in line 3-15, all the enabled configurations from current configuration $c = (P,V)$ will be returned (line 16). Regarding checking of condition A3 (line 9), there are two kind of actions that might not be *ϕ-invisible*, which are actions that contain publication events or actions that involved the update of global variables. Consider the metronome example, if we are checking property like whether $!\mathit{tick}$ event can be executed infinitely often, an action a with $a.\mathit{Event} = !\mathit{tick}$ is not *ϕ-invisible*. Another example is when we are checking whether $\mathit{tickNum} < 0$ is true in all situations, where *tickNum* is a global variable, an action a with $a.\mathit{EnableSiteType} = \mathit{GUpdate}$ is not *ϕ-invisible*.

3.3 Soundness

Lemma 1. *Given any two actions a_1 and a_2 in the system, and let s_1 and s_2 be the enable sites of actions a_1 and a_2 respectively. If sites s_1 and s_2 are not descendant processes of the RHS process of some pruning combinators and state objects of sites s_1 and s_2 are disjoint, then action a_1 is independent of action a_2.*

Proof. Actions a_1 and a_2 are dependent only when (a) action a_1 could disable action a_2 or vice versa or (b) starting from the same configuration, transitions a_1a_2 and a_2a_1 could result in different configurations. Situation (a) could happen if site s_1 could possibly modify the state object of site s_2 or vice versa, or when sites s_1 and s_2 are the descendant processes of the RHS process of some pruning combinators. For the latter

```
procedure $CAmple(P, V, Visited)$
    $P.Amples := \varnothing$;
    foreach $sP \in enableSubProcs(P)$ do                                // A2'(2)
        $\underline{fillAmpleRec(sP, V)}$;
        $\underline{composeAmples(P, sP, V)}$;
        foreach $ample \in P.Amples$ do
            $validAmple := true$;
            foreach $config \in ample$ do
                if $\neg$ $config$ $satisfies$ $A3$                           // A3
                  $\vee$ $config \in Visited$                                 // A4
                then
                    $validAmple := false$;
                    $break$;
            if $validAmple$ then
                $return$ $ample$;
    $return$ $Enable((P, V))$;
procedure $fillAmpleRec(P, V)$
    $P.Amples := \{\{config : Enable((P, V))$
                 $\mid \underline{checkA2Local(config)}\}\}$;                // A2'(1)
    if $P$ $is$ $terminal$ $process$ then
        $\underline{composeAmples(P, P, V)}$;
    else
        foreach $sP \in enableSubProcs(P)$ do
            $\underline{fillAmpleRec(sP, V)}$;
            $\underline{composeAmples(P, sP, V)}$;
procedure $composeAmples(P, sP, V)$
    $P.Amples := P.Amples \cup reformAmples(sP.Amples, P)$;
    $P.Amples := P.Amples \setminus \{\varnothing\}$;                         // A1
procedure $checkA2Local(config)$
    $return(config.EnableSiteType$ $is$ $stateless$ $\vee$
            $config.EnableSiteType$ $is$ $stateful$ $\wedge$
            $isPrivate(config.EnableSiteId))$ ;
```

Algorithm 1: CAmple

case, consider $x < x < (s_1 \mid s_2)$, if site s_1 published a value, site s_2 will be disabled immediately. Nevertheless, this case is ruled out by the assumption. Condition (b) could happen when sites s_1 and s_2 contain a common state object which they may modify and depend on. Therefore, conditions (a) and (b) are the results of having a common state object between sites s_1 and s_2. This implies that if sites s_1 and s_2 have disjoint state objects, actions a_1 and a_2 are independent to each other. □**end**.

Lemma 2. *Given a configuration $c = (P, V)$, and process P_1 as a descendant process of P. If P_1 is not a descendant process of the LHS process of some sequential combinators, then a site S that is private in $P_1[P]$, is permanently private in $P_1[c]$ as well.*

Proof. We prove by inspecting each rule in the operational semantics of *Orc* [29]. Only rule SEQ1V of operational semantics of *Orc* is possible to transfer the site reference from a process p to other processes, while retaining process p. Consider HCPs under rule SEQ1V in Figure 6, a site S that is private in $P_1[P_0]$ may not be private in $P_1[P_2]$, since P_3 might have the access to the reference of site S. Therefore, if we exclude this situation by assuming P_1 is not a descendant process of the LHS process of some sequential combinators, we prove the lemma. □ **end.**

We define several notions here. Given a configuration $c_g = (P_g, V_g)$, and P_d as a descendant process of P_g, with associated configuration $c_d = (P_d, V_d)$. $\mathbb{C}_{c_g}$ is defined as the set of configurations reachable by c_g in LTS; $\mathbb{P}_{c_g}$ is defined as $\{P \mid c = (P, V) \wedge c \in \mathbb{C}_{c_g}\}$; $HCP(\mathbb{P}_{c_g})$ is defined as the HCPs for each global process in $\mathbb{P}_{c_g}$; $\mathbb{H}_{c_g}$ is defined as the union of processes within each HCP in $HCP(\mathbb{P}_{c_g})$; $\mathbb{H}_{c_g}[P_d]$ is the set of processes that contain process P_d and its corresponding descendant processes in respective HCPs in $HCP(\mathbb{P}_{c_g})$, and $\mathbb{H}_{c_g}[P_d] \subseteq \mathbb{H}_{c_g}$.

Lemma 3. *If an action $a \in Act(c_d)$ satisfies A2' then the action is independent of any action $b \in Act(c')$, where $c' = (P', V')$, such that $P' = \mathbb{H}_{c_g}/\mathbb{H}_{c_g}[P_d]$, and V' is any valuation.*

Proof. Assume an action $a \in Act(c_d)$ satisfies A2', and assume the action is dependent to an action $b \in Act(c')$. Let sites s_a and s_b be the enable sites of actions a and b respectively. By A2'(1), site s_a is a stateless site or stateful site that is private in $p_a[P_g]$. Site s_a could not be a stateless site since a stateless site does not have a state object, and thus action a is trivially independent to any actions in the system by Lemma 1 and A2'(2). Therefore, site s_a is a stateful site that is private in $p_a[P_g]$. By Lemma 2 and A2'(2), site s_a is also permanently private in $p_a[c_g]$. By definition, state objects of site s_a and s_b are disjoint. By Lemma 1 and A2'(2), actions a and b are independent, a contradiction. □ **end.**

Theorem 1. *If any action $a \in Act(c_d)$ satisfies A2', then $AmpleAct(c_g) = Act(c_d)$ satisfies A2 for all traces in TS_{c_g}.*

Proof. Assume any action $a \in Act(c_d)$ satisfies A2', and $AmpleAct(c_g) = Act(c_d)$ does not satisfies A2 for some traces in TS_{c_g}. This means that there exists a finite execution fragment $l = c \xrightarrow{a_1} c_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} c_n \xrightarrow{a_{n+1}} \ldots$,where actions $a_1, \ldots, a_n \notin Act(c_d)$ and action a_{n+1} depends on some actions in $AmpleAct(c_g) = Act(c_d)$. Since Lemma 3 holds, action a_{n+1} must be from $PAct(c_d)/Act(c_d)$, we denote the enable site of action a_{n+1} as S_{n+1}. Since site S_{n+1} is disabled initially in c_d, it means that it is enabled later by a site call from a process $p' \in \mathbb{H}_{c_g}/\mathbb{H}_{c_g}[P_d]$. For sites in process P_d, site calls from a process $p' \in \mathbb{H}_{c_g}/\mathbb{H}_{c_g}[P_d]$ could only enable the sites that are shared in $p_d[P'_g]$, where P'_g is the global process of p'. We denote the set of state objects of the sites that are shared in $p_d[P'_g]$ as $\mathbb{D}_{share}$, and state object of S_{n+1} is in $\mathbb{D}_{share}$. On the other hand, by Lemma 2 and A2'(2), any action $a \in Act(c_d)$ is enabled by a site that is permanently private in

$p_a[c_g]$. By definition, state object of the enable site of any action $a \in Act(c_d)$ must not be found in $\mathbb{D}_{share}$. Therefore, action a_{n+1} is independent to all actions in $Act(c_d)$ by Lemma 1 and A2'(2), a contradiction. □ **end**.

Theorem 2. *Algorithm CAmple is sound.*

Proof. To show the soundness of the algorithm, we need to show that the returned ample set satisfies conditions A1-A4. Checking of condition A1 is done at line 28. Conditions A3 and A4 are checked at the global process level (line 9, 10) at *CAmple* since they are only concerned with the property of global process configurations, i.e. whether their actions are *ϕ-invisible* and whether they have been visited before. By Theorem 1, satisfaction of condition A2' leads to satisfaction of condition A2. Condition A2'(1) is checked at line 19. Condition A2'(2) is guaranteed by constraining the procedure *enableSubProcs*(P) (line 3) not to return LHS process of a sequential process and RHS process of a pruning process. □ **end**.

4 Evaluation

Our approach has been realized in the ORC Module of Process Analysis Toolkit (PAT) [1]. PAT is designed for systematic validation of distributed/concurrent systems using state-of-the-art model checking techniques [25,26]. It can be considered as a framework for manufacturing model-checkers. The data are obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 4GB RAM. ORC module supports verification of deadlock-freeness and Linear Temporal Logic (LTL) [24] property base on [21]. In Table 1 (A), three situations are compared: *CPOR* is the scenario where Compositional POR approach as described in Section 3 is applied; *POR* is the scenario where the classic approach of POR that only considered the concurrency of processes at level 1 is applied; *No POR/CPOR* is the scenario where neither POR nor CPOR is applied. In the table, ✓ and ✗ means the property is satisfied and violated respectively. The results are omitted (shown as "-") for states and times, if it takes more than eight hours for verification.

Model *Concurrent Quicksort* is a variant of the classic quicksort algorithm and emphasizes its concurrent perspective, as described in [18]. For model *Concurrent Quicksort*, *size* denotes the number of elements in the array to be sorted. Property (1.1) is used to verify whether elements in the array will eventually be sorted, and once sorted, it will remain sorted. Model *Readers-Writers Problem* is a famous computer science problem as described in [9], for which *size* denotes the number of readers. Property (2.1) verifies whether the model is possible to reach a state that violates the mutual exclusion condition. Model *Auction Management* is the case study in [2] which includes the use of external services. Please refer to [27] for the details of modeling external services in our work. Property (3.1) is used to verify that if an item has a bid on it, it will eventually be sold; Property (3.2) is used to verify that every item is always sold to a unique winner. Part (B) is the comparison of the effectiveness of our model checker for Orc and that of the model checker Maude [3,4]. Figures for number of rewrites and time usage for Maude model checker are from [4], which was run under 2.0GHz dual-core node with 4GB of memory. The experiments show that CPOR provides greater-scale reduction than classic POR for HCPs. In addition, our implementation with CPOR is more efficient than Maude [3,4].

Table 1. Performance evaluation on model checking *Orc*'s model

(A) Comparing difference POR methods

				States			Time(s)		
Model	**Property**	**Size**		**CPOR**	**POR**	**No POR/CPOR**	**CPOR**	**POR**	**No POR/CPOR**
Concurrent Quicksort	(1.1)	2	✓	58	1532	10594	0.08	1.13	5
		3	✓	69	3611	36794	0.11	8.48	74
		5	✓	237	-	-	0.68	-	-
Readers-Writers Problem	(2.1)	2	✗	106	1645	7620	0.07	1.12	4
		3	✗	152	18247	142540	0.11	14.86	101
		10	✗	472	-	-	0.49	-	-
Auction Management	(3.1)	N.A.	✓	869	-	-	0.6	-	-
	(3.2)	N.A.	✓	883	-	-	0.75	-	-

(B) Comparing Our Model Checker and Maude

			States/Rewrites		Time(s)	
Model	**Property**		**Our**	**Maude**	**Our**	**Maude**
Auction Management	(3.1)	✓	869	7052663	0.6	14.4
	(3.2)	✓	883	8613539	0.75	19.8

5 Related Work

This work is related to research on applying POR to hierarchical concurrent systems. Lang et al. [20], proposed a variant of POR using compositional confluence detection. The proposed method works by analyzing the transitions of the individual process graphs as well as the synchronization structure to identify the confluent transitions in the system graph. Transitions within the individual process graphs (at level 1) are assumed to be dependent, thus all possible transitions will be generated for individual process graphs. While in our work, we further exploit the independency within each process recursively. Basten et al. [6], proposed an approach to enhance POR via process clustering. The proposed method combines processes (at level 1) in clusters, and applies partial order reduction at proper cluster-level to achieve more reduction. Krimm et al. [19], proposed an approach to compose the processes (at level 1) of an asynchronous communicating system incrementally, and at the same time apply POR for the generated LTS. Both approaches of [6] and [19] have the assumption that the local transitions of each process (at level 1) are dependent. To the best of the author's knowledge, there is no existing work that applies POR in the context for HCP. The reason for not including orthogonal approaches such as [20,6,19] for comparisons in Section 4 is because they optimized POR by restructuring or leveraging the information of processes at level 1, while CPOR is aimed to extend POR for HCP. This means that they could be similarly used to optimize CPOR, in the same way they are used to optimize classic POR.

This work is also related to research on verifying *Orc*. Liu et al. [10], proposed an approach to translate the *Orc* language to Timed Automata, and use model checker like UPPAAL for verification. However, no reduction is considered. Alturki et al. [2,3], proposed an approach to translate the *Orc* language to rewriting logic for verification. An operational semantics of *Orc* in rewriting logic is defined, which is proved to be semantically equivalent to the operational semantics of *Orc*. To make the formal analysis more efficient, a reduction semantics of *Orc* in rewriting logic is further defined, which is proved to be semantically equivalent to the operational semantics of *Orc* in rewriting logic. We have compared the efficiency of our model checker with theirs in Section 4.

6 Conclusion

In this paper, we proposed a new method, called Compositional Partial Order Reduction (CPOR), which aims to provide the reduction with a greater scale than current partial order reduction methods in the context of hierarchical concurrent processes. It has been used in model checking *Orc* programs. Experiment results show that CPOR provide significant state-reduction for *Orc* programs. There are many languages other than *Orc* that could have the structure of HCP such as process algebra languages (e.g. CSP [14]) or service orchestration languages (e.g. BPEL [16]). Similar to classic POR method, the main challenge of applying CPOR for a language is to find an appropriate local criteria of A2 for that language. In addition, Algorithm 1 in the paper needs to be adjusted according to the semantics of the specific language. As for future works, we would further evaluate CPOR by applying it for verifying programs in other languages.

References

1. PAT: Process Analysis Toolkit, `http://www.comp.nus.edu.sg/~pat/research/`
2. AlTurki, M., Meseguer, J.: Real-time rewriting semantics of orc. In: PPDP, pp. 131–142 (2007)
3. AlTurki, M., Meseguer, J.: Reduction semantics and formal analysis of orc programs. Electr. Notes Theor. Comput. Sci. 200(3), 25–41 (2008)
4. AlTurki, M., Meseguer, J.: Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis. Technical report, The University of Illinois at Urbana-Champaign (April 2010), `https://www.ideals.illinois.edu/handle/2142/15414`
5. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge (2007)
6. Basten, T., Bosnacki, D.: Enhancing partial-order reduction via process clustering. In: ASE, pp. 245–253 (2001)
7. Clarke, E.M., Filkorn, T., Jha, S.: Exploiting Symmetry In Temporal Logic Model Checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 450–462. Springer, Heidelberg (1993)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000)
9. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM 14(10), 667–668 (1971)
10. Dong, J.S., Liu, Y., Sun, J., Zhang, X.: Verification of computation orchestration via timed automata. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 226–245. Springer, Heidelberg (2006)
11. Emerson, E.A., Sistla, A.P.: Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. ACM Transactions on Programming Languages and Systems (TOPLAS) 19(4), 617–638 (1997)

12. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
13. Håkansson, J., Pettersson, P.: Partial order reduction for verification of real-time components. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 211–226. Springer, Heidelberg (2007)
14. Hoare, C.A.R.: Communicating Sequential Processes. International Series on Computer Science. Prentice-Hall, Englewood Cliffs (1985)
15. Holzmann, G.J.: On-the-fly model checking. ACM Comput. Surv. 28(4es), 120 (1996)
16. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language Version 2.0. (April 2007), `http://www.oasis-open.org/specs/#wsbpelv2.0`
17. Kitchin, D., Quark, A., Cook, W., Misra, J.: The orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
18. Kitchin, D., Quark, A., Misra, J.: Quicksort: Combining concurrency, recursion, and mutable data structures. Technical report, The University of Texas at Austin, Department of Computer Sciences
19. Krimm, J.-P., Mounier, L.: Compositional state space generation with partial order reductions for asynchronous communicating systems. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 266–282. Springer, Heidelberg (2000)
20. Lang, F., Mateescu, R.: Partial order reductions using compositional confluence detection. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 157–172. Springer, Heidelberg (2009)
21. Liu, Y.: Model Checking Concurrent and Real-time Systems: the PAT Approach. PhD thesis, National University of Singapore (2010)
22. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
23. Peled, D.: Ten years of partial order reduction. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 17–28. Springer, Heidelberg (1998)
24. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM 32(3), 733–749 (1985)
25. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
26. Sun, J., Liu, Y., Roychoudhury, A., Liu, S., Dong, J.S.: Fair model checking with process counter abstraction. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 123–139. Springer, Heidelberg (2009)
27. Tan, T.H., Liu, Y., Sun, J., Dong, J.S.: Compositional Partial Order Reduction for Model Checking Concurrent Systems. Technical report, National Univ. of Singapore (April 2011), `http://www.comp.nus.edu.sg/pat/fm/cpor/CPORTR.pdf`
28. Valmari, A.: The state explosion problem. In: Petri Nets, pp. 429–528 (1996)
29. Wehrman, I., Kitchin, D., Cook, W., Misra, J.: A timed semantics of orc. Theoretical Computer Science 402(2-3), 234–248 (2008)