Partial Order Reduction for Timed Actors

Maryam Bagheri[1(✉)], Marjan Sirjani[2], Ehsan Khamespanah[3], Hossein Hojjat[3,4], and Ali Movaghar[1]

[1] Sharif University of Technology, Tehran, Iran
mbagheri@ce.sharif.edu
[2] Mälardalen University, Västeras, Sweden
[3] University of Tehran, Tehran, Iran
[4] Tehran Institute for Advanced Studies, Tehran, Iran

Abstract. We propose a compositional approach for the Partial Order Reduction (POR) in the state space generation of asynchronous timed actors. We define the concept of *independent actors* as the actors that do not send messages to a common actor. The approach avoids exploring unnecessary interleaving of executions of independent actors. It performs on a component-based model where actors from different components, except for the actors on borders, are independent. To alleviate the effect of the cross-border messages, we enforce a *delay condition*, ensuring that an actor introduces a delay in its execution before sending a message across the border of its component. Within each time unit, our technique generates the state space of each individual component by taking its received messages into account. It then composes the state spaces of all components. We prove that our POR approach preserves the properties defined on timed states (states where the only outgoing transition shows the progress of time). We generate the state space of a case study in the domain of air traffic control systems based on the proposed POR. The results on our benchmarks illustrate that our POR method, on average, reduces the time and memory consumption by 76 and 34%, respectively.

Keywords: Actor model · Partial order reduction · Composition · Verification

1 Introduction

Actor [1,14] is a mathematical model of concurrent computations. As units of computation, actors have single threads of execution and communicate via asynchronous message passing. Different variants of actors are emerged to form the concurrent model of modern programming languages, e.g. Erlang [26], Scala [12], Akka [2], Lingua Franca [16], and simulation and verification tools, e.g. Ptolemy [20] and Afra [21]. In the interleaving semantics of actors, the executions of actors are interleaved with each other. State space explosion is a fundamental problem in the model checking of actors. Interleaving of actor executions results in a huge state space and henceforth exponential growth in the verification time.

R. Bloem et al. (Eds.): NSV 2021/VSTTE 2021, LNCS 13124, pp. 43–60, 2022.
https://doi.org/10.1007/978-3-030-95561-8_4

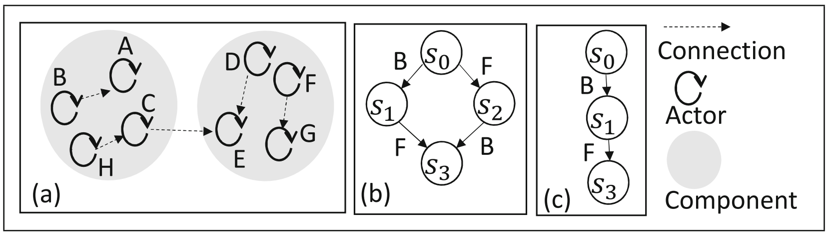

Fig. 1. (a): An actor model with two components. Connections show how actors communicate. (b): Different interleavings of executions of two independent actors B and F. State space of (b) is reduced to (c) using POR.

Partial Order Reduction (POR) [10,18] is a well-known technique to tackle the state space explosion problem. While generating the state space, POR avoids the exploration of unnecessary interleaving of independent transitions.

In this paper, we propose a compositional approach for POR of timed actors. We describe our approach on Timed Rebeca [22,24]. Actors in Timed Rebeca can model the computation time or the communication delays in time-critical systems. Standard semantics of Timed Rebeca is based on the Timed Transition System (TTS) [15]. TTS has instantaneous transitions over which the time does not progress (the so-called discrete transitions) and the timed transitions that model the progress of time. In this semantics, there is a notion of logical time that is a global time synchronized between the actors. The instantaneous transitions model executions of the actors and are interleaved if more than one actor is executed at each logical time. The time progresses if no instantaneous transition is enabled. We call a state whose outgoing transition is a timed transition a timed state.

Our POR method works on an actor model where actors are grouped together as components. We define the concept of *independent actors* as the actors that do not send messages to a common actor. Actors from different components, except for the actors on the borders of the components, are independent. We show that we can abstract the interleaved executions of independent actors within one time unit while preserving all the properties on the timed states. Dependent actors sending messages to a common actor within the same logical time are the cause of different ordering of messages in the message queue of the common actor and hence different future execution paths. The set of actors in a component are dependent. The actors sitting on the border of a component may communicate via cross-border messages to actors in other components. For such cross-border messages, we enforce a *delay condition* in this paper. The delay condition forces an actor to introduce a delay in its execution before sending a cross-border message. This way, we can avoid the interleaved executions of actors from different components in the current logical time and postpone any simultaneous arrival of messages to a common actor on borders to the next logical time. We introduce *interface components* to send such messages in the next logical time. Our method performs two operations at each logical time to generate the state space: first, builds the state space of each component, and second, composes the state spaces. We call the TTS built using our method the Compositionally-built TTS (C-TTS). Figure 1(a) shows an actor model with two components where the independent actors B and F are triggered at a logical time. As Fig. 1(b) shows, two sequences of transitions $s_0 \xrightarrow{B} s_1 \xrightarrow{F} s_3$ and $s_0 \xrightarrow{F} s_2 \xrightarrow{B} s_3$ are different interleavings of executions of B and F, both reaching the state s_3. With respect to the system properties, only one of these interleavings is necessary. Using our method, each final state in the state space of a component at the current logical time, e.g., s_1, is the initial state to generate the state space of a second component at the current logical time (Fig. 1(c)). Each final state in the state space of the second component at the current logical time is the initial state to generate the state space of a third one, and so on, no matter how the components are ordered to generate their state spaces.

An actor can send a message across the boundary of its component, and this message can interfere with another message if both messages are sent to a common actor at the same logical time. Our POR method is only applicable if an actor introduces a delay in its execution before sending a message across the border of its component. This way, there is no need to interleave executions of independent actors from different components. Let actor H in Fig. 1(a) sends a message to actor C at a logical time. In response, C is triggered but does not send a message to E at the current logical time, and it can only send a message to E in the next logical time (or later). However, in the next logical time, the message sent by C to E may interfere with a message sent to E by D (which belongs to the same component as E). Our method is aware of communications of actors over different components. For each component, we define an interface component that simulates the behaviors of the environment of the component by sending messages to the component while generating its state space.

We prove that our method preserves the properties over timed states. We reduce TTS (built by the standard semantics) and C-TTS by abstracting and removing all instantaneous transitions, and prove that the reduced transition systems are isomorphic. To investigate the efficiency of our method, we use a case study from the air traffic control systems.

Related Work. To apply standard POR techniques to timed automata, [6] proposes a new symbolic local-time semantics for a network of timed automata. The paper [11] adopts this semantics and proposes a new POR method in which the structure of the model guides the calculation of the ample set. In [17], the author proposes a POR method for timed automata, where the method preserves linear-time temporal logic properties. The authors of [13] introduce an abstraction to relax some timing constraints of the system, and then define a variant of the stubborn set method for reducing the state space. Compared to our method, none of the above approaches are compositional. In [25], there is a compositional POR method for hierarchical concurrent processes. The ample set of a process in the orchestration language *Orc* is obtained by composing ample sets of its subprocesses. Compared to [25], our method, instead of dynamically calculating the ample sets, uses the static structure of the model to remove the unnecessary interleavings.

There are several approaches for formal specification and analysis of actor models, i.e., Real-time Maude [27] is used for specifying and statistical model checking of composite actors [9], and McErlang [8] is a model checker for the Erlang programming language. The compositional verification of Rebeca is proposed in [24], but it does not perform on timed actors. To the best of our knowledge, none of them proposes a compositional approach for POR of timed actors in generating the state space.

Contributions. The contributions of the paper are as follows:

1. We propose a compositional approach for POR of timed actors. Using independent actors and the assumption of delay conditions for cross-border message passing, this approach reduces the time and memory consumption in generating state spaces by removing redundant interleavings in executions of actors.
2. We prove that our POR method preserves properties over timed states. The proof reduces TTS and C-TTS and shows that their reduced versions are isomorphic.

Our method performs on a component-based actor model, so, in the case of having a flat model, we need to organize actors into several groups of ideally independent actors, or use actors that have a delay before sending a message to determine the borders between different components. As future work, we plan to perform static analysis of models for grouping actors as components.

2 Background: Timed Rebeca

Timed Rebeca is a timed version of Rebeca [24] created as an imperative interpretation of the actor model [7]. In Timed Rebeca, communication is asynchronous, and actors have message bags to store their incoming messages that are tagged with their arrival times. Each actor takes a message with the least arrival time from its bag and executes the corresponding method, called message server. In message servers, an actor can send messages to its known actors and change values of its state variables. The actor executes the method in an atomic and non-preemptive way. Each actor can use the keyword *delay* to model passing of time during the execution of the method. In Timed Rebeca, the keywords *delay* and *after* are used to enforce the increase of logical time. To simplify the description of the method, we only consider *delay*.

In the standard semantics of Timed Rebeca, the logical time is a global time synchronized between actors. The only notion of time in our method is the logical time, so hereafter, we use the term "time" and "logical time" interchangeably. To simplify the description of the method, we assume that actors in this paper only have one message server, so, we present the simplified standard semantics of Timed Rebeca in this section.

Formal Specification of Timed Rebeca. A Timed Rebeca model $\mathcal{M} = \|_{j \in AId}\, a_j$ consists of actors $\{a_j | j \in AId\}$ concurrently executing, where AId is the set of the identifiers of all actors. An actor a_j is defined as a tuple (V_j, msv_j, K_j), where V_j is the set of all state variables of a_j, msv_j is the message server of a_j, and K_j is the set of all known actors of a_j.

Simplified Standard Semantics of Timed Rebeca. The standard semantics of $\mathcal{M}$ is the TTS $T = (S, s_0, Act, \rightarrow, AP, L)$, where S is the set of states, s_0 is the initial state, Act is the set of actions, $\rightarrow \subseteq S \times (Act \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation, AP is the set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labeling function associating a set of atomic propositions with each state. The state $s \in S$ consists of the local states of all actors along with the current time of the state. The local state of an actor a_j is $s_{a_j} = (v_j, B_j, res_j, pc_j)$, where v_j is the valuation of the state variables, B_j is the message bag storing a finite sequence of messages, $res_j \in \mathbb{R}_{\geq 0}$ is the resuming time, and $pc_j \in (\mathbb{N} \cup \{0\})$ is the program counter referring to the next statement after completing the execution of *delay*. In the message $m_k = (vals_k, ar_k)$ with the unique identifier k, $vals_k$ is a sequence of values and ar_k is the arrival time. Let S_{a_j} be the set of all states of the actor a_j. The set S is defined as $\mathbb{R}_{\geq 0} \times \prod_{j \in AId} S_{a_j}$, where $\prod$ is the Cartesian product. So, the state $s \in S$ is $(now_s, Atrs_s)$, where now_s is the current time in s and $Atrs_s$ contains the states of all actors. In s_0, each actor a_j has an initial message in its bag, and res_j, pc_j, and now_{s_0} are zero.

The set of actions is defined as $Act = Msg \cup \{\tau_j | j \in AId\}$, where Msg is the set of all messages. The transition relation $\rightarrow$ contains the following transitions that are related to taking a message and triggering an actor, resuming the execution of an actor, and progressing the time.

1 − ***Message taking transition***. (s, m_k, s'), $m_k \in Msg$, iff in the state s there is an actor a_j such that $m_k = (vals_k, ar_k)$ is a message in B_j, $ar_k \leq now_s$, and res_j is zero. The state s' results from the state s through the atomic execution of two activities: m_k is removed from B_j, and the message server of a_j is executed. The latter may change the local state of a_j and send messages that are tagged with the arrival time now_s and are stored in bags of the receiver actors. If a_j executes *delay*, the execution of the actor is suspended, the sum of now_s and the introduced delay value is stored in res_j, and pc_j is set to the location of the statement after the executed *delay*.

2 − ***Internal transition***. (s, τ_j, s') iff in the state s there is an actor a_j such that $res_j > 0$ and $res_j = now_s$. The state s' results from s by resuming the execution of the message server of a_j from the location referred to by pc_j. This case may add messages to actors' bags and change the state of a_j. Besides, res_j and pc_j are set to zero unless *delay* is executed.

3 − ***Time progress transition***. (s, d, s'), $d \in \mathbb{R}_{\geq 0}$, iff there is an actor a_j such that $res_j \neq 0$, and for each actor a_k, either B_k is empty or $res_k > now_s$. The state s' results from s through progressing the time. The time progresses to the smallest time of all resuming times that are greater than zero. The amount of the time progress is denoted by d.

The *message taking* and *internal* transitions are instantaneous transitions over which the time does not progress. These transitions have priority over the *time progress* transition; the third transition is only enabled when no transitions from the other two types are enabled. The *time progress* transition is also called a timed transition. An actor may show different behaviors depending on the order in which the messages are taken from its bag.

Timed Rebeca Extended with Components. Our POR method performs on a component-based Timed Rebeca model which consists of a set CO of components $\{co_i | i \in CID\}$, where CID is the set of all component identifiers. A component $co_i = \|_{j \in AId_i}\ a_j$ encapsulates a set of actors $\{a_j | j \in AId_i\}$, where AId_i is the set of identifiers of all actors in co_i, and $AId = \bigcup_{i \in CID} AId_i$. A local state of co_i consists of the local states of its actors and is an instance of $S_{co_i} = \prod_{j \in AId_i} S_{a_j}$. So, the state $s \in S$ is an instance of $\mathbb{R}_{\geq 0} \times \prod_{i \in CID} S_{co_i}$ and is defined as $(now_s, CAtrs_s)$, where now_s is the current time in s and $CAtrs_s$ contains the states of all components.

3 Overview of the Proposed POR Method

At each logical time, our POR method iterates over the components. To avoid exploring interleavings of executions of *independent actors* from different components, it generates the set of reachable states of each component using the *message taking* and *internal* transitions, and composes the sets of reachable states. Using the *time progress* transition, the time progresses, and the same procedure repeats for the newly generated states. Our method should be aware of the messages sent to the component while generating its state space. The order in which these messages are sent to the component may affect the reachable state. Below, we define *interface components* that are responsible for sending such messages to the components. We also describe the *delay condition* making our method applicable and explain the method using an example.

Modeling Interface Components. An actor of a component may send messages across the component's border. An actor with this ability is called a *boundary actor*. All actors of a component except for the boundary actors are called *internal actors*. The message sent by a boundary actor across the border of its component interferes with another message if both messages are sent at the same time to the same actor. The order in which these messages are taken from the bag of the receiver actor affects the system state. So we need to consider the interleavings of executions of actors of a component and its neighboring actors. A neighboring actor of a component co_i is a boundary actor of co_j, if this actor can directly communicate with an actor in co_i. To make the components independent while considering the mentioned interleavings, our method defines an *interface component* for each component. An interface component of co_i, denoted by $co_{int,i}$, contains a set of actors called interface neighboring actors. Corresponding to each neighboring actor a_j of co_i, an interface neighboring actor with the same behaviors as a_j is defined in $co_{int,i}$. Instead of neighboring actors, interface neighboring actors in $co_{int,i}$ are triggered or resume their executions to send messages to co_i. To generate the state space of co_i in isolation, executions of actors of co_i and $co_{int,i}$ are interleaved with each other.

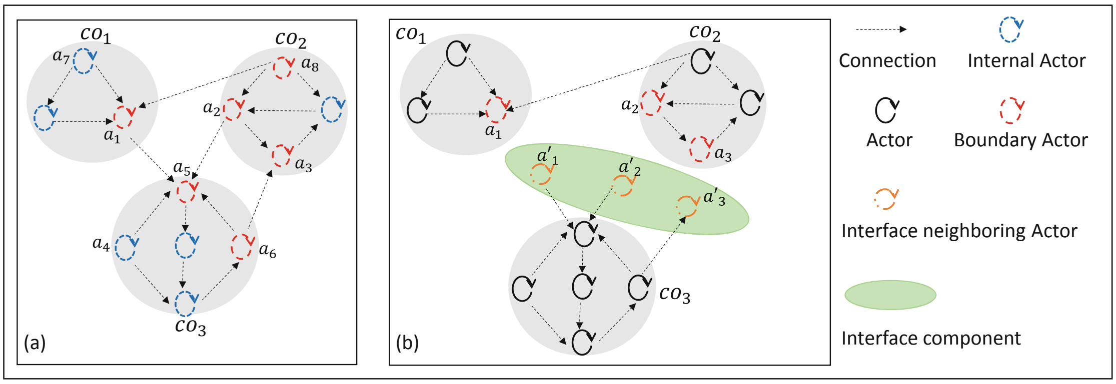

Fig. 2. (a): Three components co_1, co_2, co_3. (b): Interface component of co_3.

Figure 2(a) shows an actor model with three components co_1, co_2, and co_3. The internal and boundary actors of each component are respectively shown in blue (dotted rounded arrow) and red (dashed rounded arrow). The actors a_1, a_2, and a_3 are neighboring actors of co_3. Let actors a_1, a_2, a_3, a_4, and a_6 in Fig. 2(a) send messages to actor a_5 at the current time. The order in which these actors send their messages is important. The interface component of co_3, shown in Fig. 2(b), contains the interface neighboring actors of co_3, i.e. actors a'_1, a'_2, and a'_3 that respectively correspond to actors a_1, a_2, and a_3. The same as the neighboring actors, interface neighboring actors can communicate with boundary actors of co_3.

The Delay Condition. When an actor of a component co_i sends a message to a boundary actor of co_i, in response, the boundary actor is triggered and may send a message across the component's border. Therefore, an internal actor may be the source of interferences between messages. In such a case, interleaving executions of internal actors of two components has to be considered. For instance, let actors a_1 and a_2 in Fig. 2(a), by respectively taking a message from actors a_7 and a_8 at the current time, be triggered, and in response, send a message to actor a_5. Actor a_5 may receive the message of actor a_1 first if actor a_7 is triggered first, and may receive the message of actor a_2 first if actor a_8 is triggered first. Therefore, interleaving executions of actors a_8 and a_7 is important. To reduce interferences between messages and have independent transitions, we consider that a boundary actor is not able to send a message across the border of its component unless it has introduced a delay greater than zero before sending the message. So, actors a_1 and a_2 do not send a message to actor a_5 at the same time they receive a message. Using this condition, interleaving the executions of internal actors of two components (independent actors) is not needed. This condition is not out of touch with reality, because this delay can be the communication delay or the computation time.

We describe the POR method using Fig. 3. The right side shows three components and the left side shows how the state space of the model is generated. We divide each state into three parts, where each part denotes the local state of a component and its interface component. We show the new local state of each

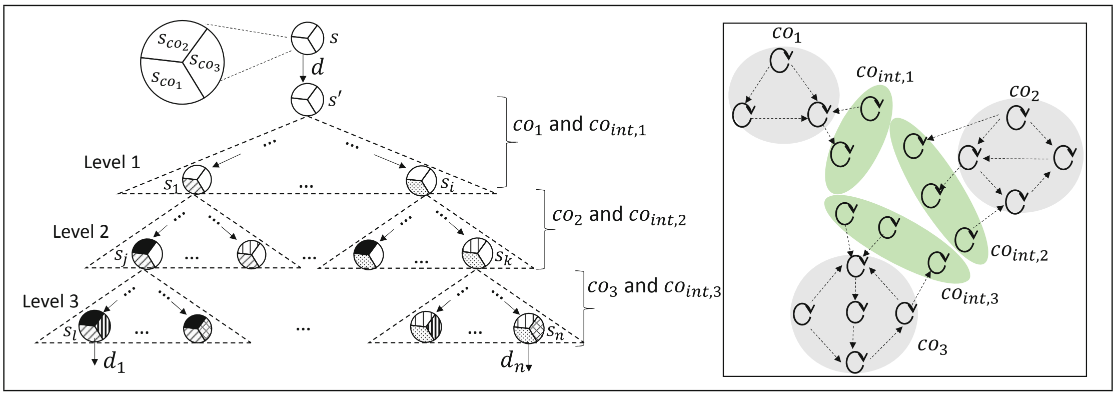

Fig. 3. The left side shows how the state space of the model of the right side is generated using the compositional approach. The interface component of co_i is denoted by $co_{int,i}$.

component with a different color. Since the state of an interface component does not contribute to verifying a property, we remove it from the figure. Let the state s be a timed state at which the time progresses. At the current time of the state s', the method generates the state space of co_1, considering the messages sent to it by $co_{int,1}$. In this case, the local states of co_2 and co_3 do not change. The new local states of co_1 are shown with different colors in states $\{s_1, \cdots, s_i\}$.

Each state of $\{s_1, \cdots, s_i\}$ is an initial state from which the state space of co_2, considering $co_{int,2}$, is generated. The method generates the state space of co_2 only once and copies the built state space for each state $s_v \in \{s_1, \cdots, s_i\}$. Then, the method updates states of the state space copied for s_v such that the local states of all components except for co_2 are set to their values in s_v. For instance, the most left triangle in the second level of Fig. 3 shows that only the local state of co_2 has changed, while the local states of co_1 and co_3 have the same values as the state s_1. Similarly, for each state of $\{s_j, \cdots, s_k\}$, the state space of co_3 is created. Finally, the time progresses at each state of $\{s_l, \cdots, s_n\}$, and the same procedure repeats. In the next section, we present the algorithm of the method.

4 The POR Algorithm

Figure 4 shows the high-level pseudo-code of our POR method. The function *porMethod* progresses the time and invokes *createInStateSpace* to generate the state space at the current time. The function *createInStateSpace* invokes *executeCOM* to generate the state space of a component considering its interface component. For instance, *porMethod* progresses the time in s in Fig. 3 and then invokes *createInStateSpace*. This function generates the whole state space from s' to $s_l, \cdots, s_n$ in three iterations. It generates each level of Fig. 3 in one iteration where it invokes *executeCOM* only once to generate a triangle and copies and updates the triangle several times.

Let $queue_{timed}$ in *porMethod* be a queue of timed states. The algorithm uses the *deQueue* function to take the head of $queue_{timed}$ (line 6) and calls *timeProg* (line 7). The *timeProg* function progresses the time and returns s' and d as the

```
Function porMethod (CO, s_0)
Input: CO set of components whose boundary actors follow the delay
        condition, s_0 the initial state
Output: S, T sets of states & transitions
(S_timed, newS, newT) ← createInStateSpace(CO, s_0)
queue_timed ← ⟨S_timed⟩
S ← {s_0} ∪ newS, T ← newT
while queue_timed ≠ ∅ do
    s ← deQueue(queue_timed)
    (s′, d) ← timeProg(s)
    S ← S ∪ {s′}, T ← T ∪ {(s, d, s′)}
    (S_timed, newS, newT) ← createInStateSpace(CO, s′)
    queue_timed ← ⟨queue_timed | S_timed⟩
    S ← S ∪ newS, T ← T ∪ newT
return (S, T)
Function createInStateSpace (CO, s)
Input: CO set of components, s a state
Output: S_frontier, S sets of states, T set of transitions
S_frontier ← {s}, S ← ∅, T ← ∅
updateIntComp(s, CO)
foreach co ∈ CO do
    (st, trans, finalSt) ← executeCOM(co, s)
    leavesOfaCom ← ∅
    while S_frontier ≠ ∅ do
        s′ ← take(S_frontier)
        (newS, newTr, newFS) ← updateSts(CO, co, s′, st, trans, finalSt)
        leavesOfaCom ← leavesOfaCom ∪ newFS
        T ← T ∪ newTr, S ← S ∪ (newS ∪ newFS)
    S_frontier ← leavesOfaCom
return (S_frontier, S, T)
Function executeCOM (co, s)
Input: co a component, s a state
Output: S_in, T sets of states & transitions, leavesOfCom final states of the
         state space of co
leavesOfCom ← ∅, S_in ← ∅, T ← ∅
enabledActors ← getEnabledActors(s, co)
if enabledActors = ∅ then
    return (∅, ∅, {s})
while enabledActors ≠ ∅ do
    (aid, msg) ← take(enabledActors)
    s′ ← trigger(s, aid, msg)
    if msg = null then
        T ← T ∪ {(s, τ_aid, s′)}
    else
        T ← T ∪ {(s, msg, s′)}
    (newS, newTr, newFS) ← executeCOM(co, s′)
    leavesOfCom ← leavesOfCom ∪ newFS
    T ← T ∪ newTr
    if newFS ≠ {s′} then
        S_in ← S_in ∪ newS ∪ {s′}
return (S_in, T, leavesOfCom)
```

Fig. 4. State-space generation by the compositional approach

new state and the amount of the time progress based on which the state space is updated (line 8). Then, the algorithm invokes *createInStateSpace* to generate the state space at the current time (line 9). This function returns the set of timed states (leaves), the set of states, and the set of transitions of the state space. The timed states are added to the end of $queue_{timed}$ (line 10), over which *porMethod* repeats the same process (line 5). Based on the semantics described in Sect. 2, the initial state s_0 is not a timed state. The function handles s_0 as a separate case; without progressing the time, uses *createInStateSpace* to generates the state space at time zero (line 2).

Let $S_{frontier}$, including the given state s in *createInStateSpace* (line 15), stores final states of state spaces generated for a component from different initial states. For instance, it stores states $\{s_1, \cdots, s_i\}$ in the first level or states $\{s_j, \cdots, s_k\}$ in the second level of Fig. 3. Assume a_{id} is a neighboring actor and a'_{id} is the interface neighboring actor corresponding to a_{id}, where id is an arbitrary index. The algorithm first uses the function *updateIntComp(s,CO)* to update states of interface components of all components (line 16). Using this function, the local state of each interface neighboring actor, i.e. a'_{id}, is updated to the local state of the corresponding neighboring actor, i.e. a_{id}, in s. The variables, the resuming time, and the program counter of the interface neighboring actor a'_{id} are respectively set to values of the variables, the resuming time, and the program counter of the corresponding neighboring actor a_{id} in s. Then, the algorithm iterates over components (lines 17 to 25) and performs as follows. For each component co, it uses the function *executeCOM* to generate the state space of co from the given base state s (line 18). This function returns the states of the state space that are not final states, the transitions, and the final states of the state space. The algorithm then iterates over $S_{frontier}$ (lines 20 to 24). It uses the *take* function to take a state s' from $S_{frontier}$ (line 21) and uses the function *updateSts* to make a copy from the generated state space and update the states of the copy one based on s' (line 22); except for the state of the component co, states of all components are set to their values in s'. The final states (leaves) of the copied state space are stored in *leavesOfaCom* (line 23), and all created states and transitions are stored (line 24). When for each state of $S_{frontier}$ as an initial state, the state space of co is built, $S_{frontier}$ is updated to *leavesOfaCom* (line 25). The final states of the state spaces of the last component are timed states that are returned.

The function *executeCOM* in Fig. 4 has a recursive algorithm that uses depth first search to generate the state space of a given component from a given state considering only the *message taking* and *internal* transitions. This function interleaves executions of actors and the interface neighboring actors of the component. The function *getEnabledActors* returns a set of tuples (aid, msg) where aid is the identifier of an actor or an interface neighboring actor of the component and msg is a message or a null value (line 30). This function returns (aid, msg) with $msg \neq null$ if actor a_{aid} can take the message msg from its bag in the state s (the *message taking* transition) and returns (aid, msg) with $msg = null$ if a_{aid} can resume its execution in s (the *internal* transition). The algorithm then iteratively takes a tuple (line 34). The algorithm triggers the actor or resumes

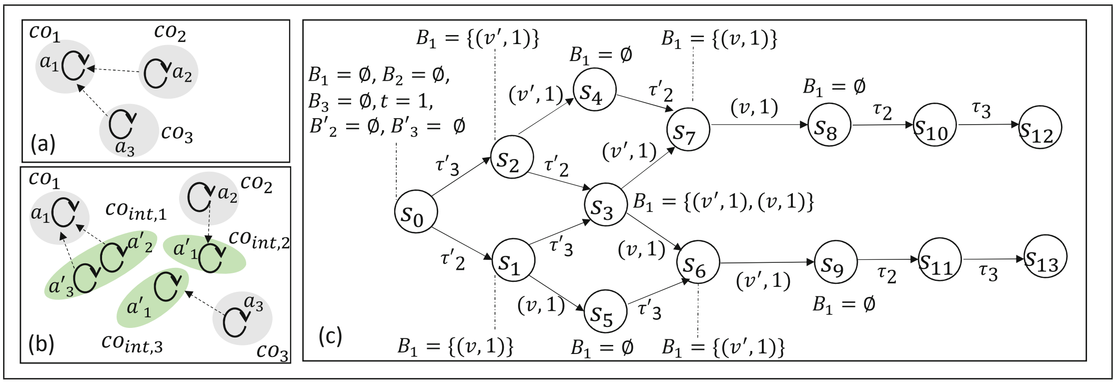

Fig. 5. (a),(b): Three components and their interface components. (c): State space in $t = 1$. B_j, B'_j bags of a_j and a'_j, respectively.

its execution using the function *trigger* (line 35). As a result, a new state is generated, and a transition is added to the set of transitions (lines 36 to 39). Then, the algorithm is executed for the new state (line 40). It stores the final states of the state space of the component (line 41) and the states that are not final states (line 44) in two disjoint sets. Finally, the transitions and the states of the state space of the component are returned (line 46). It is worthy to mention that messages sent by a component co_i to a component co_j are not stored, because these messages are generated for co_j using $co_{int,j}$.

We use the example in Fig. 5 to describe our approach. Figure 5(a) shows an actor model with three components, Fig. 5(b) shows the interface component of each component, and Fig. 5(c) shows the state space of the model for time $t = 1$. We assume that a_2 and a_3 resume their executions at time $t = 1$ and respectively send v and v' as the sequences of values to actor a_1. For $j = 1, 2, 3$, we use B_j and B'_j to denote the bag of actor a_j and the bag of the interface neighboring actor a'_j, respectively. We also use τ_j and τ'_j to denote the internal transitions over which a_j and a'_j resume their executions, respectively. The actor a_1 takes a message and performs a computation. The algorithm generates the state spaces of co_1, co_2, and co_3 in order. To generate the state space of co_1, instead of actors a_2 and a_3, actors a'_2 and a'_3 resume their executions to send messages to actor a_1. The time in all states is 1. To have a simple figure, we do not label the states with state variables of the actors. The label of each state only shows how the bag of an actor is changed when the actor is triggered or the execution of another actor is resumed. For instance, over the transition from s_0 to s_2, B_1 changes to $\{(v', 1)\}$ and the bags of other actors remain unchanged. The actors a_2 and a_3 are respectively triggered in the states s_9 and s_{11} (s_8 and s_{10}) and values of their state variables are stored.

5 Correctness Proof

We prove that our POR method preserves the properties with state formulas over timed states. We reduce TTS and C-TTS by removing all instantaneous transitions and prove that the reduced TTS and the reduced C-TTS are isomorphic. A similar reduction is used (with no proof) in [23] by Sirjani et al., where they show how a hardware platform can be used to hide from the observer the interleaved execution of a set of events (instantaneous transitions) occurring at the same logical time.

We prove that our POR method preserves deadlock: if there is a deadlock state, TTS and C-TTS reach it at the same logical time. Let $T = (S, s_0, Act, \rightarrow, AP, L)$ be the transition system of a component-based timed Rebeca model. The set S contains two sets of states: timed states and instantaneous states. The only enabled transition of a timed state is a timed transition and all outgoing transitions from an instantaneous state are instantaneous transitions.

Definition 1. *(Timed state) $s \in S$ in a given T is a timed state if there exists a state $s' \in S$ and a value $d \in \mathbb{R}_{>0}$ such that $(s, d, s') \in \rightarrow$.* □

Definition 2. *(Instantaneous state) $s \in S$ in a given T is an instantaneous state if there exists a state $s' \in S$ and an action $act \in Act$ such that $(s, act, s') \in \rightarrow$.* □

According to the standard semantics in Sect. 2, the sets of timed states and instantaneous states are disjoint since a timed transition is enabled in the state which does not have an enabled instantaneous transition. The set S contains a deadlock state if deadlock happens in the system. A state with no outgoing transition is a deadlock state.

Definition 3. *(Deadlock state) $s \in S$ in T is a deadlock state if there is no state $s' \in S$ and $l \in (Act \cup \mathbb{R}_{>0})$ such that $(s, l, s') \in \rightarrow$.* □

To simplify the proofs of this section, we add a dummy state to the set S and define a dummy transition as a timed transition with an infinite value between a deadlock state and the dummy state. If $s \xrightarrow{d} s'$ is a dummy transition where s is a deadlock state, s' is the dummy state, and d is infinite. The dummy state has no outgoing transition. Let $T_{TTS} = (S_1, s_0, Act, \rightarrow_1, AP, L)$ and $T_{CTTS} = (S_2, s_0, Act, \rightarrow_2, AP, L)$ be respectively TTS and C-TTS of a component-based Timed Rebeca model. We use $(v_j^s, B_j^s, res_j^s, pc_j^s)$ to denote the local state of the actor a_j in a state s.

Definition 4. *(Relation between a State of TTS and a State of C-TTS). A state $s \in S_1$ and a state $s' \in S_2$ are in the relation $\mathcal{R} \subseteq S_1 \times S_2$ if and only if s and s' are equal, which means:*

- *$now_s = now_{s'}$,*
- *$\forall i \in CID, \forall j \in AId_i,\ v_j^s = v_j^{s'},\ B_j^s = B_j^{s'},\ res_j^s = res_j^{s'}$, and $pc_j^s = pc_j^{s'}$.* □

Let $e = s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \cdots s_{n-1} \xrightarrow{l_{n-1}} s_n$ be an execution path from a given state s_1 to a reachable state s_n, where for all $x \in [1, n-1]$, $l_x \in (Act \cup \mathbb{R}_{>0})$. Having the relation $\mathcal{R}$ between two states s and s', i.e. $(s, s') \in \mathcal{R}$, we are able to prove that all executions from s and s' reach the same set of timed states. Note that by defining a dummy transition, a deadlock state is also a timed state.

Lemma 1. *Let* $(s_1, s'_1) \in \mathcal{R}$ *and for all* $x \in [1, n-1]$*,* $act_x \in Act$ *and for all* $y \in [1, n'-1]$*,* $act'_y \in Act$ *. For each execution* $e = s_1 \xrightarrow{act_1} s_2 \xrightarrow{act_2} \cdots \xrightarrow{act_{n-1}} s_n \xrightarrow{d} t$ *in* T_{TTS}*, there is an execution* $e' = s'_1 \xrightarrow{act'_1} s'_2 \xrightarrow{act'_2} \cdots \xrightarrow{act'_{n'-1}} s'_{n'} \xrightarrow{d} t'$ *in* T_{CTTS} *such that* $(s_n, s'_{n'}) \in \mathcal{R}$*, and vice versa.*

Proof. This proof consists of two parts:

***Part1*: For each execution e in T_{TTS} there is an execution e' in T_{CTTS} such that** $(s_n, s'_{n'}) \in \mathcal{R}$. Let $BA_{co_i} = \{a_j | \ j \in AId_i \ \wedge \ \exists z \in K_j \cdot \exists i' \in CID \cdot (i \neq i' \ \wedge \ z \in AId_{i'})\}$ be the set of boundary actors of the component co_i. Assume that a boundary actor a_j is triggered at the current time of s_1, i.e., $\exists x \in [1, n-1] \cdot act_x = m_k \ \wedge \ m_k \in B_j^{s_x} \ \wedge a_j \in BA_{co_i}$. Based on the input of the function *porMethod* in Fig. 4, a_j follows the delay condition and does not send a cross-border message at the current time, and hence, interleaving executions of internal actors of different components does not affect the reachable timed state. Let $seq_{co_i,e}$ contains the messages and the orders in which those massages are taken from the bags of actors of the component co_i over e, i.e., if $seq_{co_i,e} = \langle m_{k_1,1}, m_{k_2,2}, \cdots, m_{k_w,w}\rangle$, then $\forall j, z \in [1, w] \cdot z > j \cdot \ \exists act_x, act_{x'} \cdot x, x' \in [1, n-1] \wedge act_x = m_{k_j,j} \wedge act_{x'} = m_{k_z,z} \wedge x' > x \wedge m_{k_j,j} \in B_b^{s_x} \wedge m_{k_z,z} \in B_p^{s_{x'}} \wedge b, p \in AId_i$.

Similarly, $seq_{co_i,e'}$ can be defined for an execution e' in T_{CTTS}. If for the execution e in T_{TTS} there exists an execution e' in T_{CTTS} such that for all $i \in CID$, $seq_{co_i,e} = seq_{co_i,e'}$, then e' reaches a state $s'_{n'}$ where $(s_n, s'_{n'}) \in \mathcal{R}$. This is because the messages and the orders in which these messages are taken from the bags affects the reachable system states. We use proof by contradiction to show that there is such an execution. Assume that $(s_1, s'_1) \in \mathcal{R}$ but there is no execution e' with the mentioned condition. This means that for all executions e' in T_{CTTS}, $seq_{co_i,e} \neq seq_{co_i,e'}$ for some co_i. In our POR method, *createInStateSpace* generates the reachable states of all components at each logical time. It first uses the function *updateIntComp* to update the local state of each interface neighboring actor of each component to the local state of the corresponding neighboring actor of the component. The set of neighboring actors of co_i is $\{a_j | \ \exists i' \in CID \cdot (i \neq i' \wedge a_j \in BA_{co_{i'}} \wedge \exists z \in AId_i \cdot (z \in K_j \vee j \in K_z)\}$. The function *createInStateSpace* then invokes *executeCOM* to generate the reachable states of each component. The function *executeCOM* selects an actor from the set of enabled actors (line 34) and triggers the actor or resumes its execution. The set of enabled actors (line 30) includes actors of the component and its interface neighboring actors, where these actors can be triggered or resume their executions at the current time. Therefore, our POR method besides interleaving executions of actors of each component, interleaves executions of neighboring actors (through interface neighboring actors) and actors of a component. So the only case in which none of the executions e' corresponds to e, i.e. $seq_{co_i,e} \neq seq_{co_i,e'}$, is that $(s_1, s'_1) \notin \mathcal{R}$, that contradicts the assumption.

***Part2*: For each execution e' in T_{CTTS} there is an execution e in T_{TTS} such that** $(s_n, s'_{n'}) \in \mathcal{R}$. As mentioned before, an interleaving of executions of actors is considered in the generation of C-TTS; however, all interleavings of

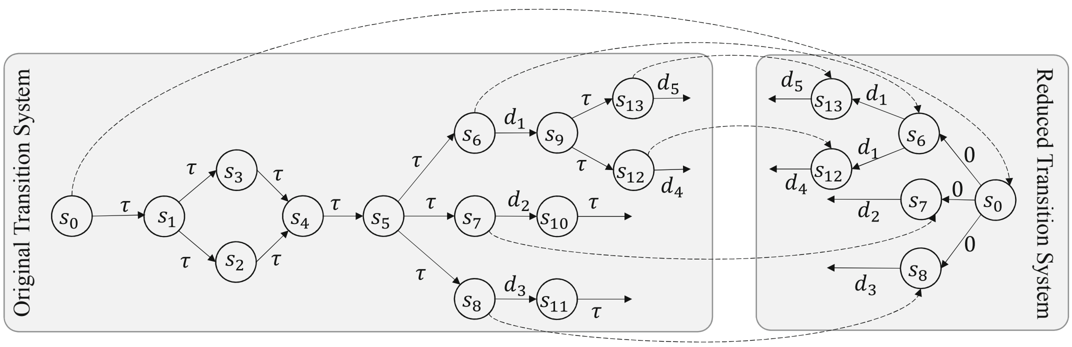

Fig. 6. The left side shows a timed transition system in which all instantaneous transitions are τ transitions and the right side shows its reduced version. The dotted arrows show the mappings between states.

executions of actors are considered in the generation of TTS. So, all reachable timed states in C-TTS can be found in TTS. □

To show that our POR method preserves the properties over timed states, we reduce TTS and C-TTS by changing the instantaneous transitions to τ transitions and removing the τ transitions. We define $Act_\tau = \{\tau\}$ and use $T_\tau = (S, s_0, Act_\tau, \rightarrow_\tau, AP, L)$ to denote a transition system in which all instantaneous transitions are changed to τ transitions.

Definition 5. *(Reduced Transition System) For $T_\tau = (S, s_0, Act_\tau, \rightarrow_\tau, AP, L)$, its reduced transition system is $T' = (S', s_0, \emptyset, \rightarrow', AP, L)$, where:*

- *$S' \subseteq S$ that contains all timed states and the state s_0,*
- *For all $s, s' \in (S' \setminus \{s_0\})$, $(s, d, s') \in \rightarrow'$ if and only if there exists an execution $s \xrightarrow{d} s_1 \xrightarrow{\tau} \ldots s_n \xrightarrow{\tau} s'$ in T_τ, where $s_1, \cdots, s_n$ are not timed states,*
- *For all $s' \in (S' \setminus \{s_0\})$, $(s_0, 0, s') \in \rightarrow'$ if and only if there exists an execution $s_0 \xrightarrow{\tau} s_1 \ldots s_n \xrightarrow{\tau} s'$ in T_τ, where $s_1, \cdots, s_n$ are not timed states.* □

There is a transition between two states of T' if and only if those are consecutive timed states or are the initial state and its following timed states in T_τ (or T). The reduced version of a transition system is shown in Fig. 6. In the following theorem, we prove that the reduced TTS and the reduced C-TTS have the same sets of states and transitions and so are isomorphic.

Theorem 1. *The reduced TTS and the reduced C-TTS are isomorphic.*

Proof. Let $T'_{TTS} = (S'_1, s_0, Act, \rightarrow'_1, AP, L)$ and $T'_{CTTS} = (S'_2, s_0, Act, \rightarrow'_2, AP, L)$ be respectively the reduced versions of TTS and C-TTS. We have $(s_0, s_0) \in \mathcal{R}$. Now, let $(s_1, s_2) \in \mathcal{R}$ and $s_1 \xrightarrow{d}{}'_1 t_1$. Based on Lemma 1, all executions from s_1 and s_2 reach the same set of timed states in TTS and C-TTS. Based on Definition 5, the set of states in the reduced TTS and the reduced C-TTS includes timed states. Therefore, there is $t_2 \in S'_2$ such that $s_2 \xrightarrow{d}{}'_2 t_2$ and $(t_1, t_2) \in \mathcal{R}$. Therefore, T'_{TTS} and T'_{CTTS} have the same sets of states and transitions, and hence, are isomorphic.

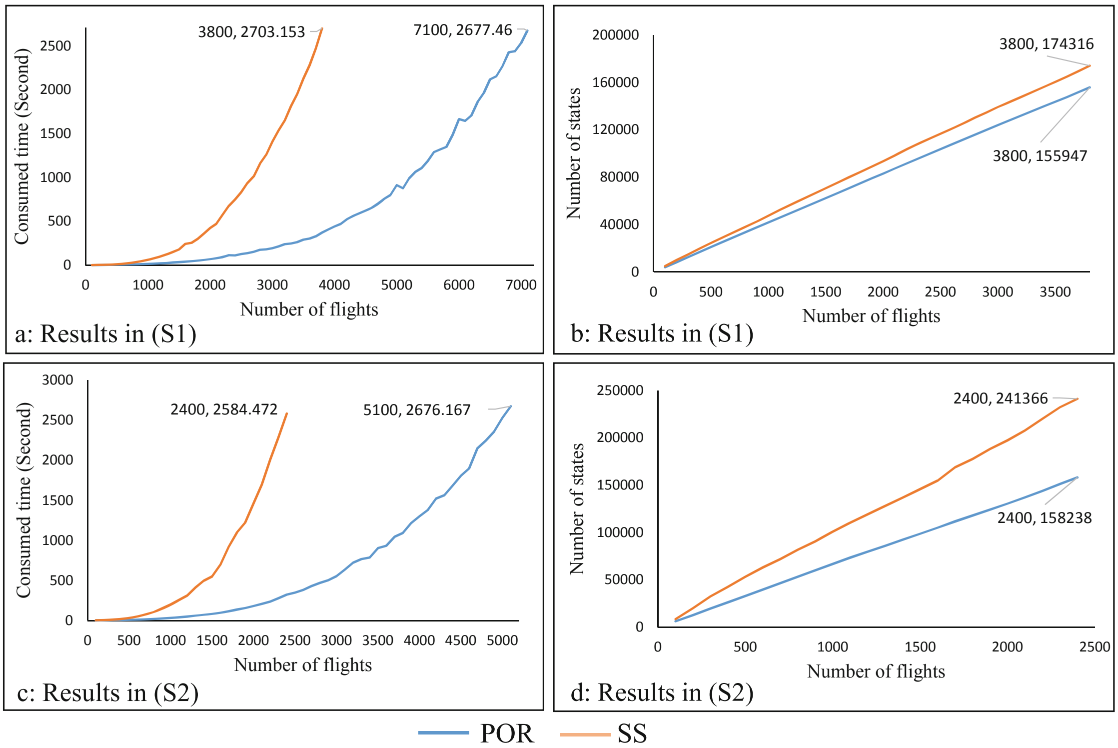

Fig. 7. The number of states and time consumption of model checking in (S1) and (S2), where "SS" stands for the standard semantics and "POR" for the POR method.

6 Experimental Results

In this section, we report our experiments on a benchmark in the domain of air traffic control systems (ATCs) [3–5] to compare the model checking time and memory consumption of using the standard semantics and the proposed POR method. We model an ATC application with four components. Each component consists of $n^2/4$ actors modeling the traveling routes in the ATC application and might consist of several actors modeling the source and destination airports. Similar to [3], we use Ptolemy II as our implementation platform to generate the state space based on both approaches. Our source codes are available in [19].

We consider three scenarios in our experiments: (S1) and (S2) that respectively use a low-concurrency model with $n = 10$ and $n = 18$, and (S3) that uses a high-concurrency model with $n = 18$. We generate a batch p of flight plans for 10000 aircraft in each scenario, where aircraft are modeled as messages passed between the actors. We partition the batch p into smaller batches $p_i, 1 \leq i \leq 100$, where p_1 contains the first 100 flight plans of p, p_2 contains the first 200 flight plans of p, and so on. By increasing the number of aircraft, the concurrency contained in the model increases. Similarly, by increasing n, the number of actors involved in the analysis and subsequently the concurrency of the model increase. Compared to (S1) and (S2), the flight plans in (S3) are selected in a way that many actors can send or receive messages corresponding to the aircraft at the same time, which lead to a high-concurrency model. We use both approaches to generate the state space of the model for each batch p_i and measure the number of states and the time consumed to generate the state space. We consider a time threshold of 45 min for generating the state space.

Figure 7 shows some results from our experiments. The legend “SS” refers to executions with the standard semantics and “POR” refers to the POR method. The POR method reduces the number of states and the time consumption of generating state spaces. Increasing the number of flights and the number of actors results in increases in the concurrency of the model, and subsequently, the time consumption and the size of state spaces. Growth in “POR” is significantly lower than “SS”, which means the POR method is more efficient when concurrency of the model increases. As Fig. 7(a) shows, the standard semantics is not scalable to a model with more than 3800 flights in (S1). The state space of a model with more than 3800 flights cannot be generated based on the standard semantics in less than 45 min. The POR method generates 286427 states for a model with 7100 flights in around 45 min. Similarly, the standard semantics cannot generate the state space of the model with more than 2400 flights in (S2). The POR method generates 333,283 states for a model with 5100 aircraft. We observe that the trends of growth in time consumption of both approaches are exponential (Figs. 7(a) and 7(c)). However, compared to “POR”, the growth order of “SS” is quadratic for our case study. The results in (S2) denote that our POR method, on average, reduces the time and memory consumption by 76 and 34%, respectively. The POR method removes unnecessary execution paths. Since several execution paths may pass through a common state, the number of transitions removed is more than the number of states removed. As the time consumption mostly relates to creation of the transitions, the reduction in the time consumption is more than the reduction in the memory consumption.

The scenario (S3) examines a model with highly concurrent actors. The standard semantics is not scalable to more than 13 flights: it generates 15,280,638 states for the model with 13 flights in 45 min. The POR method scales to the model with 220 flights. It generates 412,377 states for the model with 220 flights in 45 min.

7 Conclusion

We proposed a compositional method for POR of timed actors. Instead of interleaving executions of actors of all components to generate the state space, our method iterates over components at each logical time, generates the set of reachable states of each component, and composes the sets of reachable states. By considering the communications of actors over different components, our method interleaves executions of actors and neighboring actors of each component to generate the set of reachable states. We proved that our POR method preserves the properties of our interest.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
2. Akka library (2021). http://akka.io
3. Bagheri, M., Sirjani, M., Khamespanah, E., Baier, C., Movaghar, A.: Magnifier: a compositional analysis approach for autonomous traffic control. IEEE Trans. Softw. Eng. 1 (2021). https://doi.org/10.1109/TSE.2021.3069192
4. Bagheri, M., et al.: Coordinated actors for reliable self-adaptive systems. In: Kouchnarenko, O., Khosravi, R. (eds.) FACS 2016. LNCS, vol. 10231, pp. 241–259. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57666-4_15
5. Bagheri, M., et al.: Coordinated actor model of self-adaptive track-based traffic control systems. J. Syst. Softw. **143**, 116–139 (2018)
6. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055643
7. Boer, F.D., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5) (2017). https://doi.org/10.1145/3122848
8. Earle, C.B., Fredlund, L.Å.: Verification of timed erlang programs using McErlang. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE -2012. LNCS, vol. 7273, pp. 251–267. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30793-5_16
9. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 143–160. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37635-1_9
10. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0023731
11. 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). https://doi.org/10.1007/978-3-540-75454-1_16
12. Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2), 202–220 (2009). distributed Computing Techniques
13. Hansen, H., Lin, S.-W., Liu, Y., Nguyen, T.K., Sun, J.: Diamonds are a girl's best friend: partial order reduction for timed automata with abstractions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 391–406. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_26
14. Hewitt, C.: Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. Technical report, Massachusetts Inst of Tech Cambridge ArtificiaL Intelligence Lab (1972)
15. Khamespanah, E., Sirjani, M., Sabahi Kaviani, Z., Khosravi, R., Izadi, M.J.: Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci. Comput. Program. **98**, 184–204 (2015). special Issue on Programming Based on Actors, Agents and Decentralized Control
16. Lohstroh, M., et al.: Reactors: a deterministic model for composable reactive systems. In: Chamberlain, R., Edin Grimheden, M., Taha, W. (eds.) CyPhy/WESE -2019. LNCS, vol. 11971, pp. 59–85. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41131-2_4

17. Minea, M.: Partial order reduction for model checking of timed automata. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 431–446. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48320-9_30
18. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_34
19. Source codes (2021). https://github.com/maryambagheri1989/POR/
20. Ptolemaeus, C.: System Design, Modeling, and Simulation: Using Ptolemy II. Ptolemy.org, Berkeley, CA, USA (2014)
21. Afra Tool (2021). http://rebeca-lang.org/alltools/Afra
22. Reynisson, A.H., et al.: Modelling and simulation of asynchronous real-time systems using timed rebeca. Sci. Comput. Program. **89**, 41–68 (2014). https://doi.org/10.1016/j.scico.2014.01.008, http://www.sciencedirect.com/science/article/pii/S0167642314000239, special issue on the 10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011)
23. Sirjani, M., Lee, E.A., Khamespanah, E.: Model checking software in cyberphysical systems. In: 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 1017–1026 (2020)
24. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using rebeca. Fundam. Inf. **63**(4), 385–410 (2004)
25. Tan, T.H., Liu, Y., Sun, J., Dong, J.S.: Verification of orchestration systems using compositional partial order reduction. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 98–114. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24559-6_9
26. Virding, R., Wikström, C., Williams, M., Armstrong, J.: Concurrent Programming in ERLANG, 2nd (ed.). Prentice Hall International (UK) Ltd., GBR, London (1996)
27. Ölveczky, P.C., Meseguer, J.: Real-time maude 2.1. Electron. Notes Theor. Comput. Sci. **117**, 285–314 (2005). proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004)