Abstract—This paper presents a new process algebra, called PACoR, for real-time systems which deals with resource-constrained timed behavior as an improved version of the ACSR algebra. We define PACoR as a Process Algebra of Communicating Resources which allows to express preemptiveness, urgency and resource usage over a dense-time model. The semantic interpretation of PACoR is defined in the form of a timed transition system expressing the timed behavior and dynamic creation of processes. We define a translation of PACoR systems to Parameterized Stopwatch Automata (PSA). The translation preserves the original semantics of PACoR and enables the verification of PACoR systems using symbolic model checking in UPPAAL and statistical model checking UPPAAL SMC. Finally we provide an example to illustrate system specification in PACoR, translation and verification.

I. INTRODUCTION

More and more complex systems are being used in a safety and mission critical setting. Such systems need to be specified in a rigorous and formal way and proved to be safe and correct in practice. As a complement to the many transition system based formal models, process algebras provide a means of compact and precise formal specification framework with a calculi for checking properties. This paper defines an improved dense time process algebra (PACoR) and provides a link to verification tools which makes it relevant for practical verification.

Algebra of Communicating Shared Resources (ACSR)[4, 14] is a process algebraic approach to specify resource-constrained real-time systems. It introduces the notion of prioritized transitions based on a preemption relation of concurrent real-time processes. Recently, ACSR has become more relevant as multi-core systems are increasingly being used for real-time systems. ACSR has inspired Process Algebra for Demand and Supply (PADS)[16] to specify hierarchical scheduling systems. ACSR has been defined over a dense real-time semantics, however the underlying verification tools, VERSA[7] and XVERSA[6], have only been able to deal with the discrete time version. Moreover, ACSR has some difficulties in describing real-time systems. In the continuous-time version of ACSR timed actions cannot be restored after they have been preempted [4], which is contrary to actual real-time system implementations. This notion of processes being restored can be described in the discrete time version of ACSR, but only with the granularity of one time unit.

Inspired by ACSR, this paper presents a new process algebra, called Process Algebra of Communicating Resources (PACoR), to address the above issues. PACoR supports more expressive descriptions for timed actions in terms of the urgent- ness and preemptiveness. Moreover, we present translation rules from PACoR to Parameterized Stopwatch Automata (PSA) models which can be analyzed using UPPAAL and UPPAAL SMC. The translation preserves the semantics of PACoR. The rest of this paper is organized as follows: Section II presents the related work. Section III presents the syntax and semantics of PACoR. Section IV provides the translation rules. Section V provides a train platform system example. Finally section VI concludes the paper.

II. RELATED WORK

Process algebras, like CCS [15] and CSP [12], have originally been introduced as a formal way to rigorously describe concurrent and communicating systems together with a calculi for the verification of their properties. Over the years, different algebras have been developed for the specification and analysis of domain-specific systems like ACSR [4], [14], mCRL2 [11] for resource sharing, Timed CSP [17] for time modeling, etc.

mCRL2 is a process algebra that includes data and time suitable for the specification and verification of real-time systems. It enables local communication, multi-actions and communication-independent parallelism that are key concepts of component-based and hierarchical systems. A toolset supporting the verification of mCRL2 models has been developed[8]. However the syntactic richness of mCRL2 makes the semantics and interaction between the different concepts hard to grasp, and leads to state space explosion that affects verification.

An alternative to algebra based formalisms are timed automata-based formalisms [2], [9], where systems are described as a set of concurrent automata modeling the concrete timed behavior of the systems. Such formalisms enable certain flexibility when describing systems but may suffer to deal with hierarchy. The PACoR language is relatively compact in terms of syntax comparing to other algebra, but provides sufficient expressiveness to model resource sharing in real-time systems in an elegant way, and also enables effective dense-time analysis thanks to its timed semantics and to the UPPAAL toolsuite.

The research presented in this paper has been partially supported by EU Artemis Projects CRAFTERS and MBAT.
III. PACoR: PROCESS ALGEBRA OF COMMUNICATING RESOURCES

Process Algebra of Communicating Resources (PACoR) is a revised and improved version of ACSR for requirement specification of real-time systems. It adopts the concept of timed actions and event actions of ACSR. The execution of a timed action requires a set of resources (potentially empty) that are requested with individual priorities. The execution time is given as an interval between best and worst cases. Event actions are instantaneous, and two events (e.g., e and ¨e) synchronize if they are compatible. An internal action, represented by the distinguished event τ, cannot synchronize with any other action. Unlike ACSR, we have chosen in PACoR not to associate priorities to the event actions. The reason for this being that event actions are not resource consuming, while our language framework is resource based. Moreover, PACoR changes the meaning of the ACSR operator "\{\}" so that timed actions can be non-urgent and preemptive. Instead of \{\}, we introduce (\) as a urgent non-preemptive operator by which the execution of timed actions is never allowed to stop once it starts. The urgency can be loosened or hardened when the operators (\) and (\} are composed with the scope operator Δ.

A resource requirement is specified as a pair (r, pri) representing a resource name r and a set of events. We use natural numbers; x, y, z \in \mathbb{N} for time values; and \delta \in \mathbb{R} as a (potentially empty) sequence of resource requirements. A resource requirement is specified with a timed action (\delta) binds the timed action A to a temporal scope and incorporates time-out and exception handlers. For a given time bound n \in \mathbb{R}_+ \cup \{\infty\}, the scope may be exited in 3 ways: 1) if A successfully terminates using resources for δ time units before the deadline n, then the system proceeds to the following process; 2) if A fails to execute and misses its deadline n, then the system proceeds to a time-out handler P2; 3) the scoped timed action can be interrupted by an exception handler P1 at any time. Urgent and non-preemptive timed actions associated with the scope operator Δ can be delayed unless resources are available, but they cannot cease using resources once they start the execution. In contrast, non-urgent and preemptive timed actions are enforced to execute as soon as they start by setting the deadline as same as the execution time. The Scope constructor EΔ(n, P1, P2) for event actions binds an event action E to a temporal scope, which requires an event to happen within the temporal scope n. The process P1 executes when the event E does not happen within the temporal scope n. The waiting of event E can be interrupted by an exception process P2. The Close operator [P] associates a special resource r0 to all simultaneously enabled timed actions, so that all enabled timed actions including the empty-set timed action (\{\}) become comparable according to the priority relation. Moreover, it enables to choose a process among a set of processes using the same priorities for the same resources. The Restriction operator, P\{F\}, limits the behavior of P in a way that no event with labels in F is permitted to execute. The construct rec X : P denotes a (finite) recursive process.

A. Syntax of PACoR

The syntax of PACoR is defined by the following grammar:

\[
\begin{align*}
P & ::= \text{NIL} \mid \text{DONE} \mid A : P \mid E.P \mid P_1 + P_2 \\
& \mid P_1 || P_2 \mid [P] \mid P \\triangleright F \mid \text{recX}.P \\
A & ::= A^{\Delta}(n, P_1, P_\_2) \mid A^{\delta} \\
E & ::= E\triangleright (n, P_1, P_\_2) \mid E \\
A & ::= \{S\} \mid \langle S\rangle \\
E & ::= e \mid \bar{e} \mid \tau \\
S & ::= e \mid (r, pri), S \\
\delta & ::= [l; u]
\end{align*}
\]

NIL is a process that cannot progress, meaning that it is in deadlock. DONE is a regular termination statement. There are two prefix operators, corresponding to actions and events respectively. The timed action operator A : P executes a resource-consuming action A. A^{[l; u]} : P executes for a specific time bound \delta = [l; u], and then proceeds to the process P. The event action operator E.P executes an event action instantaneously. Basically, the event actions are urgent, but they can be delayed by the scope operator. The Choice construct P1 + P2 represents a non-deterministic choice between two processes P1 and P2. The construct P1 || P2 represents the parallel composition of two processes P1 and P2 where they may progress synchronously or independently. PACoR includes also two scope constructors to bind timed and event actions to timing requirements. A^{[l; u]}Δ(n, P_1, P_\_2) binds the timed action A to a temporal scope and incorporates time-out and exception handlers. For a given time bound n \in \mathbb{R}_+ \cup \{\infty\}, the scope may be exited in 3 ways: 1) if A successfully terminates using resources for δ time units before the deadline n, then the system proceeds to the following process; 2) if A fails to execute and misses its deadline n, then the system proceeds to a time-out handler P2; 3) the scoped timed action can be interrupted by an exception handler P1 at any time. Urgent and non-preemptive timed actions associated with the scope operator Δ can be delayed unless resources are available, but they cannot cease using resources once they start the execution. In contrast, non-urgent and preemptive timed actions are enforced to execute as soon as they start by setting the deadline as same as the execution time. The Scope constructor EΔ(n, P1, P2) for event actions binds an event action E to a temporal scope, which requires an event to happen within the temporal scope n. The process P1 executes when the event E does not happen within the temporal scope n. The waiting of event E can be interrupted by an exception process P2. The Close operator [P] associates a special resource r0 to all simultaneously enabled timed actions, so that all enabled timed actions including the empty-set timed action (\{\}) become comparable according to the priority relation. Moreover, it enables to choose a process among a set of processes using the same priorities for the same resources. The Restriction operator, P\{F\}, limits the behavior of P in a way that no event with labels in F is permitted to execute. The construct rec X : P denotes a (finite) recursive process.

### Table I

**Example: Preemptive RM task scheduling in PACoR**

<table>
<thead>
<tr>
<th>System</th>
<th>def</th>
<th>Example 3.1: Table I shows a PACoR example of two periodic tasks having the following attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>def</td>
<td>$[{D_1 \parallel D_2 \parallel T_1 \parallel T_2 }\setminus{s_1, s_2}]$</td>
</tr>
<tr>
<td>D2</td>
<td>def</td>
<td>$\emptyset^5 : s_1, D_1$</td>
</tr>
<tr>
<td>T1</td>
<td>def</td>
<td>$s_1 \triangleright (\infty, NIL, NIL, NIL).C_1$</td>
</tr>
<tr>
<td>T2</td>
<td>def</td>
<td>$s_2 \triangleright (\infty, NIL, NIL, NIL).C_2$</td>
</tr>
<tr>
<td>C1</td>
<td>def</td>
<td>${(cpu, 3)}^{[2; 2]}\Delta(5, NIL, NIL) : T_1$</td>
</tr>
<tr>
<td>C2</td>
<td>def</td>
<td>${(cpu, 2)}^{[3; 3]}\Delta(7, NIL, NIL) : T_2$</td>
</tr>
</tbody>
</table>

- **Example 3.1:** Table I shows a PACoR example of two periodic tasks having the following attributes:
  - $T_1$: Period 5, Execution Time: 2, Deadline: 5
  - $T_2$: Period 10, Execution Time: 3, Deadline: 7
Table II compares ACSR Dense-Time and PACoR in terms of expressiveness for timed actions, processes creation, and verification tools. Notice that PACoR allows the preemption of timed actions. In ACSR, a process in which a timed action using a resource is preempted cannot restore the execution of such an action. For instance, the following is a typical way where process \( P \) waits to use the resource \( cpu \) in dense-time ACSR:

\[
P \overset{\text{def}}{=} \emptyset : P \uparrow \{(cpu_1, 3)\}^5 : P'
\]

In this ACSR specification, the process \( P \) is idling until the resource \( cpu \) is available. If \( cpu \) is free, the timed action \( \{(cpu_1, 3)\}^5 \) starts its execution. However, it will be in a deadlock if the timed action can be preempted again. In contrast, PACoR specifies the non-urgent and preemptive timed actions in the following way:

\[
P \overset{\text{def}}{=} \{(cpu_1, 3)\}^5 : P'
\]

In this PACoR specification, \( P \) can either execute the timed action \( \{(cpu_1, 3)\} \) or wait for \( cpu \). PACoR allows \( P \) to restore and execute using the resource even if it is earlier preempted.

In addition to the static creation of processes, PACoR provides dynamic creation and termination of processes, adopting the new features of UPPAAL. In ACSR, a process never dies or terminates once it starts. In contrast, PACoR supports process termination, such a termination does not influence any process execution. In our framework, the static process creation can be supported by both UPPAAL and UPPAAL SMC verification techniques, but the dynamic process creation can be supported only by UPPAAL SMC.

C. Timed Operational Semantics

In the following, we define the semantics of PACoR in terms of a timed transition system (TTS). Such a semantics allows expressing both the discrete behavior of the process terms and the timed progress. Given a timed action \( A \), the function \( \rho(A) \) returns the set of resources required by \( A \). The function \( lp(A) \) states 1) the lowest priority of requests of action \( A \) if \( A \) has shared resources, 2) \( lp(A) = 0 \) if \( \rho(A) = \emptyset \), otherwise 3) \( lp(A) = 1 \) if \( A \) has no shared resources and \( \rho(A) \neq \emptyset \). Moreover, we generalize the timed action priority relation in order to consider event actions, making then all action types comparable. This priority relation also functions as a preemption relation between timed actions, and thus specifies in which cases a timed action can preempt another timed action.

**Definition 3.1: (Priority Relation)** Given two actions \( \alpha \) and \( \beta \), we say that \( \beta \) has priority over \( \alpha \), denoted by \( (\alpha < \beta) \), if one of the following conditions holds:

1. \( \alpha \in \mathcal{D}_R \) and \( \beta \in \mathcal{D}_E \)
2. Both \( \alpha \) and \( \beta \) are actions in \( \mathcal{D}_R \), where

\[
\forall r \in \rho(\beta) \cap \rho(\alpha), (r, p) \in \alpha \land (r, p') \in \beta \Rightarrow p < p'
\]

According to the first condition, all instantaneous events have priority over timed actions. Besides, timed actions sharing a resource are arbitrated according to the second condition: if there exist resources shared by two timed actions \( \alpha \) and \( \beta \), then \( \beta \) preempts \( \alpha \) if and only if all the priorities of \( \alpha \) over the shared resources are lower than the priorities of \( \beta \) over the same resources. Our preemption relation is more strict than the ACSR preemption relation but helps to solve non-deterministic relations between tasks. Such a fact makes the system more deterministic.

**Example 3.2:** some relations between two timed actions can be as follows:

1. \( \{(r_1, 2), (r_2, 5)\} \prec \{(r_1, 7), (r_2, 5)\} \)
2. \( \{(r_1, 2), (r_2, 5)\} \prec \{(r_2, 7), (r_3, 5)\} \)
3. \( \{(r_1, 2), (r_2, 5)\} \prec \{(r_1, 7), (r_2, 3)\} \)
4. \( \{(r_1, 2), (r_2, 0)\} \prec \{(r_1, 7)\} \)
5. \( \{(r_1, 2), (r_2, 1)\} \prec \{(r_1, 7)\} \)
6. \( \{(r_3, 1), (r_2, 3), (r_3, 1)\} \prec \{(r_1, 1), (r_2, 1), (r_3, 1)\} \)

In ACSR, cases 2 and 4 are incomparable, no one has priority over the other, but in case 6 actions are comparable.

Let us first recall timed transition systems and their bisimulation relation. Timed Transition Systems (TTS) [13] represent an elegant model to define the semantics of real-time formalisms. Basically, a TTS is a labeled transition system where labels can be events or durations.

**Definition 3.2 (Timed transition System):** a timed transition system over an alphabet \( \Sigma \) is a tuple \( (\mathcal{S}, \mathcal{S}^0, \rightarrow) \) where \( \mathcal{S} \) is a set of states, \( \mathcal{S}^0 \subseteq \mathcal{S} \) is the set of initial states and \( \rightarrow \subseteq \mathcal{S} \times \Sigma \cup \{\tau\} \cup \mathcal{R}_{\leq 0} \times \mathcal{S} \) is the transition relation.

Here and elsewhere, we write \( s \overset{\lambda}{\rightarrow} s' \) for any transition \( (s, \lambda, s) \in \rightarrow \). Moreover, a transition is continuous if it is labeled by a real value from \( \mathcal{R}_{\geq 0} \), otherwise the transition is discrete. TTSs are comparable via simulation and bisimulation relations. The simulation relation of TTSs establishes a mapping between their timed traces [1] where, from a common state, we check whether for each outgoing transition of the simulating system, a corresponding transition can be triggered in the simulated system.

We introduce the transition relation \( \rightarrow_{\tau} = (\tau)^{\ast} \) as a sequence of internal transitions such that for any event \( \lambda \), \( s \overset{\lambda}{\rightarrow}_{\tau} s' = s \overset{\lambda}{\rightarrow}_{\tau} s_i \overset{\lambda}{\rightarrow}_{\tau} s_j \overset{\lambda}{\rightarrow}_{\tau} s' \). Such a transition relation enables to check the weak-bisimulation relation of TTS.

**Definition 3.3 (TTS simulation):** Given 2 TTSs \( T_1 = (\mathcal{S}_1, \mathcal{S}_1^0, \rightarrow_1) \) and \( T_2 = (\mathcal{S}_2, \mathcal{S}_2^0, \rightarrow_2) \) defined on the same alphabet \( \Sigma \), \( T_1 \) simulates \( T_2 \) through a relation \( \mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2 \), denoted by \( T_1 \sqsubseteq_{\mathcal{R}} T_2 \), if \( \forall s \in \mathcal{S}_1^0 \exists s' \in \mathcal{S}_2^0 (s,s') \in \mathcal{R} \).
and for all \((s_1, s_2) \in \mathcal{R}\) if \(s_1 \xrightarrow{\lambda} s_1'\) then \(\exists s_2' \in \mathcal{S}_2 | s_2 \xrightarrow{\lambda} s_2' \wedge (s_1', s_2') \in \mathcal{R}\).

Accordingly, \(T_1\) and \(T_2\) are bisimilar through the relation \(\mathcal{R}\), denoted \(T_1 \sim_\mathcal{R} T_2\), if \(T_1 \subseteq \mathcal{R} T_2\) and \(T_2 \subseteq \mathcal{R}^{-1} T_1\).

The restriction of a TTS \((\setminus)\) enables to ignore unmatched synchronizing transitions when composing different concurrent processes. Such a function implements the operator \(\setminus F\) of PACoR.

**Definition 3.4 (TTS restriction):** The restriction of a TTS over a set of events is a TTS where transitions compatible over these events are deleted. Given a TTS \(T = \langle \mathcal{S}, \mathcal{S}^0, \rightarrow \rangle\) and a set of events \(W\), we define the restriction of \(T\) on \(W\), denoted by \(T\setminus W\), to be the TTS \(\langle \mathcal{S}, \mathcal{S}^0, \rightarrow \setminus \{ s \xrightarrow{\lambda} s' | \lambda \in W \} \rangle\).

According to the following theorem, the TTS restriction operation preserves the bisimulation relation.

**Theorem 3.1 (Bisimulation and restriction):** Let \(T_1\) and \(T_2\) be 2 TTS defined on the same alphabet \(\Sigma\) and \(\mathcal{R}\) be a simulation relation, then \(T_1 \sim_\mathcal{R} T_2 \Rightarrow T_1 \Phi \sim_\mathcal{R} T_2 \Phi\) for any \(\Phi \subseteq \Sigma\).

**Proof.** It is straightforward.

We introduce \(\text{ID}\) as a set of identifiers and \(\mathcal{P}\) as a set of processes.

- \(\text{Ident} : \mathcal{P} \rightarrow \text{ID}\) is a function associating an identifier to each process in order to distinguish between processes.
- \(\text{fresh(ID)}\) returns a free identifier from \(\text{ID}\).

**Definition 3.5 (Semantics of PACoR):** We define the semantics of PACoR in terms of a timed transition system TTS \((\mathcal{S}, \mathcal{S}^0, \rightarrow)\) where:

- \(\mathcal{S} = (\text{Stmt} \cup \text{Deadlock}) \times \mathbb{R}_{\geq 0} \times 2^{\text{ID}}\) is the set of states, \(\text{Stmt}\) represents the PACoR statements derived from the non-terminal \(P\) given in the grammar, \(\mathbb{R}_{\geq 0}\) represents the time instants, and \(\text{ID}\) is a set of process identifiers.
- \(\mathcal{S}^0 \subseteq \mathcal{S}\) is the set of initial states defined by \(\text{Stmt}^0 \times \{0\} \times \{0\}\), where \(\text{Stmt}^0\) defines a set of statements representing the initial steps of the PACoR executions, and \(\{0\}\) is the identifier of the top level process \(P\).
- \(\rightarrow \subseteq \mathcal{S} \times \{\tau\} \cup E \cup A \cup \mathbb{R}_{\geq 0} \times \mathcal{S}\).

The function \(\text{Stmt}^0\) is given in the following:

\[
\begin{align*}
\text{Stmt}^0(\text{NIL}) & = \emptyset \\
\text{Stmt}^0(E,P) & = E \\
\text{Stmt}^0(A:P) & = A \\
\text{Stmt}^0(P_1 + P_2) & = \text{Stmt}^0(P_1) \cup \text{Stmt}^0(P_2) \\
\text{Stmt}^0(P_1 || P_2) & = \text{Stmt}^0(P_1) \times \text{Stmt}^0(P_2) \\
\text{Stmt}^0([P_1]) & = \text{Stmt}^0(P) \\
\text{Stmt}^0(P,F) & = \{s | s \in \text{Stmt}^0(P) \land s \notin F\} \\
\text{Stmt}^0(\text{rec}X.P) & = \text{Stmt}^0(P)
\end{align*}
\]

The transition relation \(\rightarrow\) is the smallest relation given by the rules of Table III.

In fact, we keep constructing the function \(\text{Ident}\) on the fly, so that when executing a statement we propagate the identifier of the statement process to the resulting process \((\text{Ident}(P) := \text{Ident}(A:P))\), respectively state, in order to know at any instant the owner process of a statement. The first rule of \(\text{A-success}\) in Table III corresponds to a successful execution of an urgent timed action, whereas the second rule states the execution of a regular timed action with respect to its deadline. The urgent action \(A^{[u,u]}\) should be run immediately once the corresponding statement is reached, otherwise the system will be in a deadlock according to the second rule of \(\text{A-failure}\). Moreover, an urgent action can be preempted by event actions only. \(\text{A-failure}\) rule corresponds to the execution of an action missing its deadline, the corresponding timeout handler process \(P_t\) is triggered once the deadline is met. The \(\text{Close}\) rule corresponds to the close operator where the action \(A\) is extended with a particular resource request \((r_0, l_p(A))\) in order to make the action \(A\) comparable with other concurrent actions, and possibly leads to a priority decision in some non-deterministic cases. \(\text{A-delay}\) states a delay \(m\) of a non-urgent action when the corresponding deadline \(n\) allows \((u < n - m)\). \(\text{Preemption}\) rules state a preemption of either an urgent or a non-urgent preemptive timed action. \(\text{E-failure}\) rule is, the same as \(\text{A-success}\), for urgent \((E)\) and non-urgent \((E \setminus (n, P_t, P_u))\) events execution, where no time elapses because events execution is instantaneous. Rule \(\text{E-failure}\) corresponds to the triggering of a timeout handler process when the execution of a non-urgent event misses its deadline. Rule \(\text{Exception}\) corresponds to the triggering of an exception handler process \(P_e\) when a non-urgent event or timed action is waiting to be triggered. Rule \(\text{E-delay}\) expresses a delay of a non-urgent event while the corresponding deadline \(n\) is respected. Rules \(\text{E-ChoiceL}\) and \(\text{A-ChoiceL}\) specify a non-deterministic choice between 2 executions, where the left hand statement is taken. A synchronization of 2 compatible events is described by rule \(\text{E-sync}\). The resulting transition will be considered as an internal event \(\tau\) of the composition. \(\text{E-AsyncL}\) describes an asynchronous event of parallel composition. In rule \(\text{A-sync}\), 2 concurrent timed actions progress together if they do not require the same resources. The execution time of such a synchronization will be the maximum of the execution time of both actions. \(A_1 \cup A_2\) is the union of resource requests of both actions \(A_1\) and \(A_2\). The priority relation is not considered because actions \(A_1\) and \(A_2\) do not compete the same resources, so they are incomparable. Rule \(\text{A-async}\) corresponds to the execution of a timed action \(A_1\) having priority over the concurrent timed action \(A_2\). Rule \(\text{Event-Action}\) states that event actions have priority over timed actions. Rule \(\text{NIL}\) states a system deadlock if one of the concurrent processes reaches a \(\text{NIL}\) statement. In the rules explained above, the set of activated process identifiers \(\text{ID}\) is not updated when executing any transition because no process is created or killed on the execution of these transitions. Rule \(\text{P-destroy}\) states an on-the-fly destruction of a process once it reaches a \(\text{DONE}\) statement. The identifier of the corresponding process \((\text{Ident}(\text{DONE}))\) will be so removed from the set of active process identifiers \(\text{ID}\). For this reason, we propagated the process identifiers throughout transition rules in order to recognize the identifier of the owner process of each statement. Rule \(\text{P-create}\) expresses a dynamic creation of processes where the number of active processes in a system may differ from a state to another. The on-the-fly creation of a process [3] can be interpreted by the extension of the current system state by a new instance of such a process, where a fresh identifier is picked from the set of identifiers \(\text{ID}\) and assigned to such a process instance. Such a feature is recently integrated in the UpPAAL toolsuite [10].
TABLE III
Transition relation of PACoR semantics

<table>
<thead>
<tr>
<th>Condition</th>
<th>Transition Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-success</td>
<td>[ m \in \mathbb{R}_{\geq 0}, 1 \leq m \leq u, \text{Ident}(P) := \text{Ident}(A^{[l,u]} : P) ] [ \langle A^{[l,u]} : P, x, ID \rangle \xrightarrow{A} \langle P, x + m, ID \rangle ]</td>
</tr>
<tr>
<td>A-failure</td>
<td>[ m = 0 ] [ \langle A^{[l,u]} \Delta(n, P, p, x), x, ID \rangle \xrightarrow{A} \langle P, x, ID \rangle ]</td>
</tr>
<tr>
<td>Close</td>
<td>[ m \in \mathbb{R}_{\geq 0}, \text{Ident}(P) := \text{Ident}(A^{[l,u]} : P) ] [ \langle A^{[l,u]} \Delta(n, P, p, x), x, ID \rangle \xrightarrow{Au([v(x, p)])} \langle P, x + m, ID \rangle ]</td>
</tr>
<tr>
<td>Preemption</td>
<td>[ 0 \leq m \leq l, t' = l - m, u' = u - m ] [ \langle {a}^{[l,u]} : P, x, ID \rangle \xrightarrow{E} \langle {a}^{[l',u']} : P, x + m, ID \rangle ]</td>
</tr>
<tr>
<td>E-success</td>
<td>[ \text{Ident}(P) := \text{Ident}(E, P) ] [ \langle E, P, x, ID \rangle \xrightarrow{E} \langle P, x, ID \rangle ]</td>
</tr>
<tr>
<td>E-failure</td>
<td>[ m = 0 ] [ \langle E, P, x, ID \rangle \xrightarrow{E} \langle P, x, ID \rangle ]</td>
</tr>
<tr>
<td>A-Delay</td>
<td>[ m \in \mathbb{R}_{\geq 0}, m &lt; n, n' = n - m ] [ \langle A^{[l,u]} \Delta(n, P, p, x), x, ID \rangle \xrightarrow{A} \langle A^{[l,u]} \Delta(n', P, p, x) : P, x + m, ID \rangle ]</td>
</tr>
<tr>
<td>A-ChoiceL</td>
<td>[ \text{Ident}(P) := \text{Ident}(E, P) ] [ \langle E, P_1 + P_2, x, ID \rangle \xrightarrow{A} \langle P_1, x, ID \rangle ]</td>
</tr>
<tr>
<td>E-scan</td>
<td>[ m \in \mathbb{R}_{\geq 0}, \text{Ident}(P) := \text{Ident}(A^{[l,u]} : P) ] [ \langle A^{[l,u]} \Delta(n, P, p, x), x, ID \rangle \xrightarrow{A} \langle A^{[l,u]} \Delta(n', P, p, x) : P, x + m, ID \rangle ]</td>
</tr>
<tr>
<td>E-sync</td>
<td>[ \rho(A_1) \cap \rho(A_2) = \emptyset, m = \text{max}(A_1, A_2) ] [ \langle A_1 : P_1</td>
</tr>
<tr>
<td>Event-Action</td>
<td>[ \langle A : P_1</td>
</tr>
<tr>
<td>P-destroy</td>
<td>[ \langle \text{DONE}</td>
</tr>
<tr>
<td>P-create</td>
<td>[ \langle P, x, ID \rangle \xrightarrow{P} \langle P, x, ID \cup { \text{Ident}, \text{Ident} } ]</td>
</tr>
</tbody>
</table>

IV. TRANSLATION OF PACoR TO PSA

In order to analyze PACoR models automatically, we define a translation rewriting any PACoR description in terms of UPPAAL time automata, while UPPAAL and UPPAAL SMC tools can be applied to simulate the system execution and check the underlying properties. To this end, we consider timed automata using stopwatches [5], often called Prioritized Stopwatch Automata (PSA). Roughly speaking, stopwatches are clocks that can be stopped and resumed at any time, without reinitialization, during the system execution.

A. Parametrized Stopwatch Automata

The stopwatch concept [5] provides a sophisticated clock mechanism to measure the execution time of preemptive tasks. Such a mechanism enables to accumulate time when the action is running, and once the action is preempted the stopwatch clock makes a pause and resumes when the corresponding action resumes its execution. Given a set variables \( Varr \), by \( C(Var) = \{ x \in v | x \in Varr, v \in \mathbb{R} \} \) we define the set of variable constraints of \( Varr \) where \( \in \{ <, \leq, >, \geq, = \} \).

**Definition 4.1 (Parameterized Stopwatch Automaton):** A stopwatch automaton over a set of channels \( \Omega \) is given by a tuple \((Q, q^0, Com, C, V, V^0, \rightarrowAct, Inv)\) where \( Q \) is a set of locations, \( q^0 \in Q \) is the initial location, function \( Com : Q \rightarrow \mathbb{B} \) states whether a location is committed or not, \( C \) is a set of stopwatch clocks, \( V \) is a set of discrete variables initialized according to the valuation \( V^0 \), \( \rightarrowAct \subseteq Q \times C \cup V \times \Omega! \cup \Omega? \cup \{?\} \times \Delta \times Q \) is the transition relation where \( \Delta : \{ C \rightarrow \{0\} \} \cup \{ V \rightarrow \mathbb{R} \} \) is a set of actions that reset some clocks and update some discrete variables, \( Act : Q \times C \rightarrow \mathbb{N} \) defines the set of actions that can be performed on stopwatch clocks at each location, and \( Inv : Q \rightarrow C(C) \) associates to each location clock invariants.

We write \( q_1 \xrightarrow{g/\lambda/\alpha} q_2 \) for a transition \((q_1, g, \lambda, \alpha, q_2) \in \rightarrowAct\) outgoing from \( q_1 \) to \( q_2 \) guarded by \( g \), labeled with \( \lambda \) and performing the action \( \alpha \) on clocks and discrete variables. In fact, we have used a long right-arrow \((\rightarrow)\) in order to distinguish the transition relation of PSA from that of TTS \((\rightarrow)\). \( Act(q)(c) = v \) means that the first derivative of \( c \) at location \( q \) has value \( v \), which represents the rate of progress each one time unit. So, for any duration \( d \) the clock \( c \) progresses.
with \( v \times d \).

Therefore, a network of PSA is a set of concurrent PSA instances (processes) that can be created on the fly when it is needed. These processes are defined on the same set of channels where some clocks and variables can be shared. The dynamic creation of processes is the subject of a recent extension implemented in UPFAAL. Thus, the semantics is based on a dynamically evolving set of processes where at least one process (main) is immediately created when the system starts. For this reason, we associate to each process (template instance) an identifier \( Id \) enabling us to distinguish between the different instances of the same template. Moreover, we associate to each process a fresh variable \( \text{loc} \) of type \( Q \) stating the current location of the process. We also introduce \( f \circ g \) as the right overriding function where function \( g \) overrides function \( f \) for all elements in the intersection of their domains. For all \( z \in \text{domain}(f \circ g) \),

\[
(f \circ g)(z) = \begin{cases} g(z) & \text{if } z \in \text{domain}(g) \\ f(z) & \text{if } z \in \text{domain}(f) \setminus \text{domain}(g) \end{cases}
\]

**Definition 4.2 (Semantics of a network of PSA):** Given a network of PSA \( \langle T_0, ..., T_n \rangle \) defined on the same set of channels \( \Omega \) where some discrete variables and clocks can be shared (global). Assuming that \( T_0 \) is the main (root) template of our network. We associate the identifier \( Id_0 \) and variable \( \text{loc} \) to the first instance of \( T_0 \). The semantics of network \( \langle T_0, ..., T_n \rangle \) is given by the TTS \( \langle S, S^0, \rightarrow \rangle \) where \( S = \{ s \in \text{val}(W) | \forall i s \models \text{Inv}_{i}(\text{loc}_i) \} \) is a set of evaluations, each one satisfies the local invariants of the involved locations \( \text{loc}_i \) of the corresponding active processes. \( S^0 = \bigcup \{ V^0_1 \cup \{ C_i \rightarrow \{ 0 \} \} \} \cup \{ Id_0.\text{loc} \mapsto q^0_0 \} \). \( W = \bigcup \{ Id_i.\text{C} \} \cup \bigcup \{ Id_i.\text{V} \cup \{ \text{loc} \} \} \) is the set of discrete and continuous variables of the instances that can be created together with the newly introduced variables \( \text{loc} \). The transition relation is given by the rules of table IV.

The TTS states of the semantics are partial functions where only variables of created channels are valued. The condition \( \text{comit}(s, Id) = \forall Id' \in s \text{ comit}_{Id}(s(Id'.\text{loc})) \Rightarrow \text{comit}_{Id}(s(\text{Id.}\text{loc})) \) states that if \( s \) contains a committed location, the current location of process \( Id \) should also be committed [3].

**Rule Action** of table IV states that if there exists a process \( Id \) such that its current location \( Id.\text{loc} \) corresponds to the current reached location \( q \), and the current system state \( s \) satisfies the guard \( q \) of the PSA outgoing edge from \( q \) so the system moves to another state obtained by applying action \( a \) on state \( s \) where the current location of process \( Id \) is updated to \( q' \). Rule Sync describes a synchronization of 2 edges having compatible labels when the source locations match with the current system state \( s \), and both guards are satisfied. The resulting transition is labeled with internal event \( \tau \), where the receiver action \( a_j \) is applied after taking into account the update made by the sender action \( a_i \). Rule Create expresses the on-the-fly creation of processes. Such a feature is newly implemented in UPFAAL via statement \( \text{spawn} \). \( [T]_s \) defines an instance of template \( T \) where the template parameters are evaluated according to state \( s \). The rule associates to the newly created process a fresh identifier \( Id' \leftarrow \text{fresh}(s) \) thanks to the function \( \text{Ident} \). The new instance will be immediately initialized through function \( f\text{Init} \) given by: \( f\text{Init}(Id', T) = \{ Id'.\text{loc} \mapsto T.q^0_0, \|_{v \in T.v} \{ Id'.v \mapsto T.V^0(v) \}, \|_{c \in T.C} \{ Id'.c \mapsto 0 \} \} \). Such a function consists of initializing variables and initial location according to the instantiated template declaration, whereas clocks are initialized to 0. Over the syntactic function \( \text{exit} () \) of rule Destroy, stating that the process execution is done, UPFAAL removes such a process from the system.

**Rule Delay** states that a delay \( d \) is allowed if the invariants of the current locations \( (s(Id.\text{loc})) \) of all processes \( (Id \in s) \) are still satisfied, and no one of the current locations is committed.

The time progress function \( s + d \) updates the stopwatch clocks at state \( s \) by \( d \). In fact, \( d \) is the real amount of time units elapsed whereas the update of each clock depends on its rate, defined by function \( \text{Act} \). Formally, \( s + d = \forall Id \in s, \forall c \in Id.C c := s(Id.c) + Act(Id.\text{loc})(c) \times d \).

**B. Graphical Notations of PACOR**

The translation of PACOR to timed automata is carried out in two steps: translation of a PACOR system to a graphical PACOR (gPACOR) model, and then translation of the gPACOR model into the corresponding timed automata model.

### Table V: Graphical Notations of PACOR

<table>
<thead>
<tr>
<th>Rule</th>
<th>PACOR</th>
<th>gPACOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( p \overset{\text{def}}{=} p' )</td>
<td>( \text{[} P \text{]} \to \text{[} P' \text{]} )</td>
</tr>
<tr>
<td>2</td>
<td>( p \overset{\text{def}}{=} A[l,u] : p' )</td>
<td>( \text{[} A \text{]} : l \leq t \leq u \to \text{[} P' \text{]} )</td>
</tr>
<tr>
<td>3</td>
<td>( p \overset{\text{def}}{=} \text{\it{\emph{&amp;}}} p' )</td>
<td>( \text{[} P \text{]} \to \text{[} P' \text{]} )</td>
</tr>
<tr>
<td>4</td>
<td>( p \overset{\text{def}}{=} p_1 + p_2 )</td>
<td>( \text{[} P_1 \text{]} \to \text{[} P_2 \text{]} )</td>
</tr>
<tr>
<td>5</td>
<td>( p \overset{\text{def}}{=} \text{\it{\emph{&amp;}}} p_1 | p_2 )</td>
<td>( \text{[} P_1 \text{]} \to \text{[} P_2 \text{]} )</td>
</tr>
<tr>
<td>6</td>
<td>( p \overset{\text{def}}{=} A[l,u] \text{\it{\emph{&amp;}}} p_1, p_2 )</td>
<td>( \text{[} A \text{]} : l \leq t \leq u \to \text{[} P_1 \text{]} \to \text{[} P_2 \text{]} )</td>
</tr>
<tr>
<td>7</td>
<td>( p \overset{\text{def}}{=} \text{\it{\emph{&amp;}}} \text{\it{\emph{&amp;}}} (n, p_1, p_2) )</td>
<td>( \text{[} P_1 \text{]} \to \text{[} P_2 \text{]} )</td>
</tr>
<tr>
<td>8</td>
<td>( [P] \overset{\text{def}}{=} A[l,u] : p' )</td>
<td>( \text{[} A \text{]} : l \leq t \leq u \to \text{[} P' \text{]} )</td>
</tr>
</tbody>
</table>

Table V shows PACOR syntax and the corresponding notations in gPACOR. As shown in Rule 1, a process is represented by a box with a name and a transition arrow to the definition of the process. Timed actions are depicted as a circle, called timed action node, with resource requirements \( A \), as shown in Rule 2. For timed action statements, a clock \( x \) is introduced to specify timing requirements. Using this clock, the upper bound of the execution time is represented by an invariant.
associated to the node (oval state in Rule 2) representing the timed action, and the lower bound is a guard on the outgoing transition from that node. Event actions are represented by edges from the box \( P \), as shown in Rule 3. Rule 4 and 5 show that gPACoR processes can be composed using Choice and Parallel operators. The Choice operator is represented by a choice connector \( + \) and arrows leading to possible processes, whereas the parallel composition is represented by a dashed box. The scoped timed action node, with resource requirements \( A \), is shown in Rule 6. It has 3 outgoing transitions leading to a normal process (\( P^0 \)), a time-out handler (\( P^1 \)), and an exception handler (\( P^2 \)). A new clock \( y \) is introduced for the scoped timed action to measure the time elapsed (for deadline) since the scoped action is reached. Rule 7 shows the scoped event action node. It is almost the same as the scope timed action node. The event action node triggers an event \( E \) on the normal termination outgoing transition, and does not contain any resource-consuming requirements. Rule 8 shows a timed action node which is applied with the Close operator.

Fig. 2. gPACoR model of Example 3.1

Fig. 2 shows the gPACoR model of task \( T_1 \) and its dispatcher \( D_1 \) of Example 3.1. Based on a gPACoR model, a model of PSA can be built using the realizations of the priority relations.

C. Translation of Process Models

The PSA model of a PACoR specification is graphically similar to the gPACoR description, and simply obtained by making PSA stopwatch clocks run depending on the availability of resources that the process timed actions would use. Timed actions of gPACoR including resources and timing requirements are represented in locations of PSA using stopwatch clock variables. Event actions are executed on transitions of PSA. The stopwatch clocks associated with resources can run and stop according to the availability of resources. For the convenience, we use \( R \) for a set of resources, \( A_1, A_2, \ldots \) for resource requirements, and \( A_1, A_2, \ldots \) for timed action nodes. For resource operations, we define the following functions:

- \( S(A) \) is a resource requirement that a timed action node \( A \) requires,
- \( \rho(A) \subseteq R \) is the set of resources that a timed action node \( A \) requires,
- \( \text{Request}(A) \) is a resource request according to the resource requirement \( A \),
- \( \text{Release}(2R) \) is a function that releases a set of resources \( (2R) \),
- \( \text{Available}(2R) \) is a function stating whether a set of resources \( (2R) \) is available,
- \( \text{LockPreemp}(pid) \) sets a process \( pid \) to non preemptive process.

Table VI shows how each basic structure (transition) of gPACoR is implemented in PSA. The boxes in gPACoR represent process names and correspond to the circles in PSA. Such circles are the starting locations of the PSA model. The hexagons in both sides represent definition of actions. For a given \( P_{gPACoR} \), every PSA automaton, as shown Table VI, is given with a starting location \( P_{PSA} \). In later, all starting locations will be interpreted by the commit connector of UPPAAL because the outgoing transitions linked to the starting location of PSA should always be urgently taken. For gPACoR actions, the corresponding actions in PSA are defined according to translation rules given in this section. In fact, we only focus on the translation and implementation of gPACoR transition relation.

Rule 2-1 and 2-2 show how to translate a timed action node of gPACoR to a PSA automaton. Every single timed action node of gPACoR and its clocks is transformed to a PSA with a set of clocks. In principle, a PSA process requires a set of resources using the function \( \text{Request}(A) \), according to a resource requirement \( A \) before entering the timed action location. The PSA process should release resources, using the function \( \text{Release}(2R) \), when it exits from the timed action location. Inside a timed action node, the stopwatch \( x \) runs depending on the resource availability which is checked using \( \text{Available}(2R) \). If a process wants an exclusive (non-preemptive) use of resources, it sets itself to “non-preemptive” using \( \text{LockPreemp}(pid) \).
Rule 3 shows how gPACOR events are translated to a set of transitions into the corresponding PSA model. As shown in Rule 4, the Choice operator \( \oplus \) is replaced by the commit connector \( \ominus \) of UPPAAL. The Choice operator has alternatives of possible actions. In case of timed actions, the availability of resources should be known before the selection of an outgoing transition. Thus, as shown in Rule 4-1, all resources that the alternative timed actions require are requested, e.g. \( \text{Request}(\rho(A_1)) \), before making a choice and checked to be usable (transitions are enabled) when making the choice, e.g. \( \text{Available}(\rho(A_1)) \). The Parallel operator is realized by instantiating parallel composed processes, as shown in Rule 5.

In our translation, the processes instantiation can be static or on-the-fly. The static creation is implemented by a special event, \( \text{Spawn}[\text{pid}]! \), whereas the dynamic instantiation is supported by the UPPAAL statement \( \text{Spawn} \) together with a template name. Rule 6 shows the translation of a scoped timed action of gPACOR. According to whether the action is preemptive (\( \{S\} \)) or not (\( \langle S \rangle \)), the entering transition into the timed action location executes the function \( \text{LockPreemp}(\text{pid}) \). Notice that the resource requirement \( S(A_e) \) of the exception handler \( P_e \) is required on the transition going to the timed action node,
because the exception handler can have the same possibility to execute as the scoped timed action if it is enabled. Thus the resources requests for \( P_e \) are performed before the execution of the scoped timed actions. Similarly, in case that \( P_e \) is a composition of processes using the Choice operator, all alternative transitions are directly linked to the timed action location in order to make them possibly enabled when the timed action executes.

Rule 7-1 and 7-2 show the translation of the scoped event action. In Rule 7-1, the resources request of the exception handler \( P_e \) is performed in the same way as in the scoped timed action. Rule 7-2 describes the case where the exception handler \( P_e \) carries out an event action \( e_e \). Rule 8 shows how to apply the Close operator to PSA. According to the semantics of PACoR, if a process extended with the Close operator, it adds a special resource requirement \((r, lp(A))\) to its resource requirements for the execution of its timed actions. Rule 9 and 10 show how to realize \( NIL \) and \( DONE \) actions. The \( NIL \) action is interpreted by a deadlock. For the \( DONE \) action, the final location can be one of two types: 1) Rule 10-1 shows the execution of \( DONE \) action, using the statement \( \text{exit}() \) of UPPAAL, in the case of dynamic process creation; 2) Rule 10-2 shows a final location for the static process creation, which has an outgoing transition labeled with the process instantiation event \( \text{Spawn}[pid] \)?.

### D. Composition and Instantiation of Process Models

Individual PSA automata are composed into a networked PSA system in the following way. First, every starting location for individual PSAs is replaced by a commit connector. Second, every edge leading to a final location is redirected to the starting location of the same name. PSA templates are instantiated either by static or dynamic process creation. If dynamic process creation is used, every PSA template has an initial location as shown in Fig. 3(a). Initially, only the top level process is instantiated, and other processes are instantly created when needed. If using static process creation, all processes are created at the same time and every PSA template has an initial location as shown in Fig. 3(b), except for the top level process which follows Fig. 3(a). The actual execution of a process, in case of static instantiation, begins by the instantiation event \( \text{Spawn}[pid] \) according to the Rule 5-1. Fig. 4 shows the PSA model of Example 3.1. Notice that it has the same structure as the gPACoR model of Fig 2. All boxes have been translated into initial and committed locations with names. The incoming transitions into \( C_1 \) and \( C_2 \) execute the function \( \text{request}() \) to request \( \text{cpu} \), and the outgoing transitions from them call the function \( \text{release}() \) to release the resource.

**Theorem 4.1:** The TTS semantics restricted to \( \tau \) and time-transitions of a PACoR system is bisimilar to the TTS associated to the PACoR translation into a network of PSA.

**Proof concept.** The proof is direct and consists in checking that each TTS transition, except event transitions realized by restriction, generated by one of the rules given in Definition 3.5 can also be generated using the rules given in Definition 4.2 after applying the translation of PACoR transitions according to tables VI and VII, and vice versa. A sketch of this proof is given in the Appendix.

### V. Example: Platform Control System

In order to illustrate our framework, we present an example of a train platform system in Fig. 5. In this system, the trains enter from five incoming tracks and stop in one of two segments \((seg_1, seg_2)\) on each side of the platform. Individual tracks have priorities for a train to occupy the segments, and the priorities increase every specified time interval \((wt)\) unless the train can enter the platform. The train can stay at the platform for \( st \) time units. Each track has the minimum inter-arrival time \((mt)\) to serve a train. Variable \( j \) and \( i \) are used to index segment number and train identity, respectively. In the second line of Table VIII we use a timed action with an infinite execution time but with a deadline of \( wt \). If a segment becomes available the train can occupy that segment, otherwise the train becomes a new \( TrnReqP \) process waiting for a segment, but with a priority increased by \( k \). In the experiment shown in Table IX, we have 5 tracks and 2 or 3 segments. Over this example, we show 1) how to specify the system using PACoR, 2) how to translate the PACoR model into PSA, and 3) how to apply UPPAAL and UPPAAL SMC to reason about various characteristics of the system. Table VIII describes the platform system using PACoR. The PSA model which is translated from the same PACoR model is given in the Appendix.

#### A. Analysis using UPPAAL and UPPAAL SMC

The analysis of the train platform system has been done using two verification methods, symbolic and statistical model checking. We use UPPAAL to check with symbolic model checking that there is no deadlock in the system. With UPPAAL SMC we generate the probability distribution for the waiting time of each train.
from PACoR to PSA which preserves the semantics. The translation is not implemented yet, but it is designed to utilize new features of ÜPPAAL, such as on-the-fly process creation. Finally we have provided an example illustrating the complete process from systems specification in PACoR, translation to gPACoR, further translation to PSA and verification of system properties using ÜPPAAL and ÜPPAAL SMC.

REFERENCES