Model checking process algebra of communicating resources for real-time systems

Boudjadar, Abdeldjalil; Kim, Jin Hyun; Larsen, Kim Guldstrand; Nyman, Ulrik Mathias

Publication date:
2014

Document Version
Peer reviewed version

Model Checking Process Algebra of Communicating Resources for Real-time Systems

A.Jalil Boudjadar, Jin Hyun Kim, Kim G. Larsen, Ulrik Nyman
Institute of Computer Science, Aalborg University, Denmark
{jalil,jin,kgl,ulrik}@cs.aau.dk

***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, urgentness 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 **P**rocess **A**lgebra of **C**ommunicating **R**esources (PACoR), to address the above issues. PACoR supports more expressive descriptions for timed actions in terms of the urgentness 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 B provides a train platform system example. Finally section VI concludes the paper.

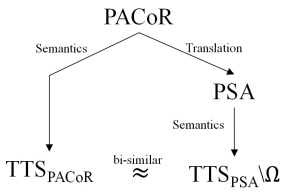


Fig. 1. Translation and Semantics

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 **A**lgebra of **Co**mmunicating **R**esources (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 $\bar{e}$) 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 $\langle\rangle$ as a **urgent non-preemptive** operator by which the execution of timed actions is never allowed to stop once it starts. The urgentness can be loosened or hardened when the operators $\langle\rangle$ and $\{\}$ are composed with the scope operator $\triangle$.

A resource requirement is specified as a pair (r, pri) representing a resource name r and a priority pri. An action is a (potentially empty) sequence of resource requirements. A timed action without any resource requirement is called empty-set timed action and denoted by $\emptyset$ or $\{\}$. An action a can be either preemptible ($\{a\}$) or non-preemptible ($\langle a\rangle$) signifying whether it can be preempted by an action having a higher priority. Higher priorities have numbers greater than that of lower priorities. In PACoR, all priorities should be greater than 0, which is reserved for the empty-set timed actions. A timed action a^δ is defined as an action a specified with a timed interval δ representing the best and worst case execution time. We use $\mathcal{D}_R$ as a set of resources and $\mathcal{D}_E$ a set of events. We also use x, y, z for clock variables; P, Q for processes; R for natural numbers; l, m, n, u for time values; and α and β for events and actions.

A. Syntax of PACoR

The syntax of PACoR is defined by the following grammar:

$$
\begin{array}{rcl}
P & ::= & NIL \mid DONE \mid A : P \mid E.P \mid P_1 + P_2 \\
 & \mid & P_1 \parallel P_2 \mid [P] \mid P\backslash F \mid recX.P \\
A & ::= & \mathcal{A}^\delta \triangle (n, P_t, P_e) \mid \mathcal{A}^\delta \\
E & ::= & \mathcal{E}\nabla(n, P_t, P_e) \mid \mathcal{E} \\
\mathcal{A} & ::= & \{S\} \mid \langle S\rangle \\
\mathcal{E} & ::= & e \mid \bar{e} \mid \tau \\
S & ::= & \epsilon \mid (r, pri), S \\
\delta & ::= & [l; u]
\end{array}
$$

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. $\mathcal{A}^{[l;u]} : P$ 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 $P_1 + P_2$ represents a non-deterministic choice between two processes P_1 and P_2. The construct $P_1 \parallel P_2$ represents the parallel composition of two processes P_1 and P_2 where they may progress synchronously or independently. PACoR includes also two scope constructors to bind timed and event actions to timing requirements. $\mathcal{A}^{[l;u]}\triangle(n, P_t, P_e)$ binds the timed action $\mathcal{A}$ 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 $\mathcal{A}$ successfully terminates using resources for δ time units before the deadline n, then the system proceeds to the following process; 2) if $\mathcal{A}$ fails to execute and misses its deadline n, then the system proceeds to a time-out handler P_t; 3) the scoped timed action can intervened by an exception handler P_e at any time. Urgent and non-preemptive timed actions associated with the scope operator $\triangle$ 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 $\mathcal{E}\nabla(n, P_t, P_e)$ for event actions binds an event action $\mathcal{E}$ to a temporal scope, which requires an event to happen within the temporal scope n. The process P_t executes when the event $\mathcal{E}$ does not happen within the temporal scope n. The waiting of event $\mathcal{E}$ can be interrupted by an exception process P_e. The Close operator $[P]$ associates a special resource r_0 to all simultaneously enabled timed actions, so that all enabled timed actions including the empty-set timed action ($\emptyset$ or $\{\}$) 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, $P\backslash F$, 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 (infinite) recursive process.

TABLE I
EXAMPLE: PREEMPTIVE RM TASK SCHEDULING IN PACOR

$$
\begin{array}{rcl}
System & \stackrel{def}{=} & [(\ D_1 \parallel D_2 \parallel T_1 \parallel T_2\ )\backslash\{s_1,\ s_2\}] \\
D_1 & \stackrel{def}{=} & \emptyset^5 : \bar{s_1}.D_1 \\
D_2 & \stackrel{def}{=} & \emptyset^{10} : \bar{s_2}.D_2 \\
T_1 & \stackrel{def}{=} & s_1\nabla(\infty, NIL, NIL).C_1 \\
T_2 & \stackrel{def}{=} & s_2\nabla(\infty, NIL, NIL).C_2 \\
C_1 & \stackrel{def}{=} & \{(cpu, 3)\}^{[2,2]}\triangle(5, NIL, NIL) : T_1 \\
C_2 & \stackrel{def}{=} & \{(cpu, 2)\}^{[3,3]}\triangle(7, NIL, NIL) : T_2
\end{array}
$$

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

D_1 and D_2 periodically dispatch T_1 and T_2, whereas C_1 and C_2 are timed actions that require the resource cpu.

B. Comparison of PACoR and ACSR Algebra

TABLE II
EXPRESSIVENESS COMPARISON OF ACSR AND PACOR

	Attribute	ACSR	PACoR
Timed Action	Non-urgent & Preemptive	✗	✓
	Urgent & Non-preemptive	✓	✓
	Non-urgent & Non-preemptive	✗	✓
Process Creation & Termination		Static	Static & Dynamic
Verification Tools		No	UPPAAL & UPPAAL SMC

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:

$$P \stackrel{def}{=} \emptyset : P \ \dagger \ \{(cpu_1, 3)\}^5 : P'$$

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:

$$P \stackrel{def}{=} \{(cpu_1, 3)\}^5 : P'$$

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 α and β, we say that β has priority over α, denoted by $(\alpha \prec \beta)$, if one of the following cases holds:

1) $\alpha \in \mathcal{D}_R$ and $\beta \in \mathcal{D}_E$
2) Both α and β are actions in $\mathcal{D}_R$, where $\forall r \in \rho(\beta) \cap \rho(\alpha),\ (r, p) \in \alpha \wedge (r, p') \in \beta \Rightarrow p < p'$

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 α and β, then β preempts α if and only if all the priorities of α over the shared resources are lower than the priorities of β 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) $\{(r_1, 2), (r_2, 5)\} \not\prec \{(r_1, 7), (r_2, 3)\}$
4) $\{(r_1, 2), (r_2, 0)\} \prec \{(r_1, 7)\}$
5) $\{(r_1, 2), (r_2, 1)\} \prec \{(r_1, 7)\}$
6) $\{(r_1, 3), (r_2, 3), (r_3, 1)\} \not\prec \{(r_1, 1), (r_2, 1), (r_3, 1)\}$

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 Σ is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \rightarrow \rangle$ where $\mathcal{S}$ is a set of states, $\mathcal{S}^0 \subseteq \mathcal{S}$ is the set of initial states and $\rightarrow \subseteq \mathcal{S} \times \Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0} \times \mathcal{S}$ is the transition relation.

Here and elsewhere, we write $s \xrightarrow{\lambda} s'$ for any transition $(s, \lambda, s) \in \rightarrow$. Moreover, a transition is continuous if it is labeled by a real value from $\mathbb{R}_{\geq 0}$, 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 $\rightarrow_\pi = (\xrightarrow{\tau})^*$ as a sequence of internal transitions such that for any event λ, $s \xrightarrow{\lambda}_\pi s' = s \xrightarrow{\tau}_\pi s_i \xrightarrow{\lambda} s_j \xrightarrow{\tau}_\pi s'$. Such a transition relation enables to check the weak-bisimulation relation of TTS.

Definition 3.3 (TTS simulation): Given 2 TTSs $T_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \rightarrow_1 \rangle$ and $T_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \rightarrow_2 \rangle$ defined on the same alphabet Σ, 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 $\forall s \in \mathcal{S}_0^1\ \exists s' \in \mathcal{S}_0^2 | (s, s') \in \mathcal{R}$

and for all $(s_1, s_2) \in \mathcal{R}$ if $s_1 \xrightarrow{\lambda} s_1'$ then $\exists s_2' \in \mathcal{S}_2 | s_2 \xrightarrow{\lambda}_\pi s_2' \ \wedge \ (s_1', s_2') \in \mathcal{R}$.
Accordingly, T_1 and T_2 are bisimilar through the relation $\mathcal{R}$, denoted $T_1 \sim_\mathcal{R} T_2$, if $T_1 \sqsubseteq_\mathcal{R} T_2$ and $T_2 \sqsubseteq_{\mathcal{R}^{-1}} T_1$.

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

Definition 3.4 (TTS restriction): The restriction of a TTS over a set of events is a TTS where transitions composable 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 $T\backslash W$, to be the TTS $\langle \mathcal{S}, \mathcal{S}^0, \rightarrow \backslash \{s \xrightarrow{\lambda} s' \ | \ \lambda \in W\} \rangle$.
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 Σ and $\mathcal{R}$ be a simulation relation, then $T_1 \sim_\mathcal{R} T_2 \Rightarrow T_1\backslash\Phi \sim_\mathcal{R} T_2\backslash\Phi$ for any $\Phi \subseteq \Sigma$.
`Proof`. It is straightforward.

We introduce **ID** as a set of identifiers and $\mathcal{P}$ as a set of processes.

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

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

- $\mathcal{S}$= $(Stem \cup \{\texttt{Deadlock}\}) \times \mathbb{R}_{\geq 0} \times 2^{\textbf{ID}}$ is the set of states, $Stem$ represents the PACoR statements derived from the non-terminal **P** given in the grammar, $\mathbb{R}_{\geq 0}$ represents the time instants, and **ID** is a set of process identifiers.
- $\mathcal{S}^0 \subseteq \mathcal{S}$ is the set of initial states defined by $Stem^0 \times \{0\} \times \{Id_0\}$, where $Stem^0$ defines a set of statements representing the initial steps of the PACoR executions, and Id_0 is the identifier of the top level process **P**.
- $\rightarrow \subseteq \mathcal{S} \times \{\{\tau\} \cup E \cup \mathcal{A} \cup \mathbb{R}_{\geq 0}\} \times \mathcal{S}$.

The function $Stem^0$ is given in the following:

$$
\begin{array}{l}
Stem^0(NIL) = \emptyset \\
Stem^0(E.P) = E \\
Stem^0(A : P) = A \\
Stem^0(P_1 + P_2) = Stem^0(P_1) \cup Stem^0(P_2) \\
Stem^0(P_1 \| P_2) = Stem^0(P_1) \times Stem^0(P_2) \\
Stem^0([P]_I) = Stem^0(P) \\
Stem^0(P\backslash F) = \{s | s \in Stem^0(P) \wedge s \notin F\} \\
Stem^0(recX.P) = Stem^0(P)
\end{array}
$$

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

In fact, we keep constructing the function $Ident$ on the fly. so that when executing a statement we propagate the identifier of the statement process to the resulting process ($Ident(P) := Ident(\mathcal{A} : P)$), respectively statement, in order to know at any instant the owner process of a statement. The first rule of **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 $\mathcal{A}^{[l,u]}$ should be run immediately once the corresponding statement is reached, otherwise the system will be in a `Deadlock` according to the second rule of **A-failure**. Moreover, an urgent action can be preempted by event actions only. **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 **Close** rule corresponds to the close operator where the action $\mathcal{A}$ is extended with a particular resource request $(r_0, lp(\mathcal{A}))$ in order to make the action $\mathcal{A}$ comparable with other concurrent actions, and possibly leads to a priority decision in some non-deterministic cases. **A-delay** states a delay m of a non-urgent action when the corresponding deadline n allows ($u < n - m$). **Preemption** rules state a preemption of either an urgent or a non-urgent *preemptive* timed action. **E-success** rule is, the same as **A-success**, for urgent ($\mathcal{E}$) and non-urgent ($\mathcal{E}\nabla(n, P_t, P_e)$) events execution, where no time elapses because events execution is instantaneous. Rule **E-failure** corresponds to the triggering of a timeout handler process when the execution of a non-urgent event misses its deadline. Rule **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 **E-delay** expresses a delay of a non-urgent event while the corresponding deadline n is respected. Rules **E-ChoiceL** and **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 **E-sync**. The resulting transition will be considered as an internal event (τ) of the composition. **E-AsyncL** describes an asynchronous event of parallel composition. In rule **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. $\mathcal{A}_1 \cup \mathcal{A}_2$ is the union of resource requests of both actions $\mathcal{A}_1$ and $\mathcal{A}_2$. The priority relation is not considered because actions $\mathcal{A}_1$ and $\mathcal{A}_2$ do not compete the same resources, so they are incomparable. Rule **A-async** corresponds to the execution of a timed action $\mathcal{A}_1$ having priority over the concurrent timed action $\mathcal{A}_2$. Rule **Event-Action** states that event actions have priority over timed actions. Rule **NIL** states a system deadlock if one of the concurrent processes reaches a NIL statement. In the rules explained above, the set of activated process identifiers ID is not updated when executing any transition because no process is created or killed on the execution of these transitions. Rule **P-destroy** states an on-the-fly destruction of a process once it reaches a $DONE$ statement. The identifier of the corresponding process ($Ident(DONE)$) will be so removed from the set of active process identifiers 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 **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 **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

$$\textbf{A-success}: \frac{m \in \mathbb{R}_{\geq 0},\ l \leq m \leq u,\ Ident(P) := Ident(\mathcal{A}^{[l,u]} : P)}{\langle \mathcal{A}^{[l,u]} : P, x, \texttt{ID}\rangle \xrightarrow{\mathcal{A}} \langle P, x+m, \texttt{ID}\rangle} \qquad \frac{m \in \mathbb{R}_{\geq 0},\ l \leq m \leq u,\ m \leq n,\ Ident(P) := Ident(\mathcal{A}^{[l,u]} \triangle(..) : P)}{\langle \mathcal{A}^{[l,u]} \triangle(n, P_t, P_e) : P, x, \texttt{ID}\rangle \xrightarrow{\mathcal{A}} \langle P, x+m, \texttt{ID}\rangle}$$

$$\textbf{A-failure}: \frac{n = 0}{\langle \mathcal{A}^{[l,u]} \triangle(n, P_t, P_e), x, \texttt{ID}\rangle \xrightarrow{\tau} \langle P_t, x, \texttt{ID}\rangle} \qquad \frac{m > u}{\langle \mathcal{A}^{[l,u]} : P, x, \texttt{ID}\rangle \xrightarrow{\mathcal{A}} \langle \texttt{Deadlock}, x+m, \texttt{ID}\rangle}$$

$$\textbf{Close}: \frac{m \in \mathbb{R}_{\geq 0},\ Ident(P) := Ident([\mathcal{A}^{[l,u]} : P]_I)}{\langle [\mathcal{A}^{[l,u]} : P]_I, x, \texttt{ID}\rangle \xrightarrow{\mathcal{A} \cup \{(r_0, lp(\mathcal{A}))\}} \langle P, x+m, \texttt{ID}\rangle} \qquad \textbf{A-Delay}: \frac{m \in \mathbb{R}_{\geq 0},\ u \leq n-m,\ n' = n-m}{\langle \mathcal{A}^{[l,u]} \triangle(n, P_t, P_e) : P, x, \texttt{ID}\rangle \xrightarrow{m} \langle \mathcal{A}^{[l,u]} \triangle(n', P_t, P_e) : P, x+m, \texttt{ID}\rangle}$$

$$\textbf{Preemption}: \frac{0 \leq m < l,\ l' = l-m,\ u' = u-m}{\langle \{a\}^{[l,u]} : P, x, \texttt{ID}\rangle \xrightarrow{a} \langle \{a\}^{[l',u']} : P, x+m, \texttt{ID}\rangle} \qquad \frac{0 \leq m < l,\ l' = l-m,\ u' = u-m,\ n' = n-m}{\langle \{a\}^{[l,u]} \triangle(n, P_t, P_e) : P, x, \texttt{ID}\rangle \xrightarrow{a} \langle \{a\}^{[l',u']} \triangle(n', P_t, P_e) : P, x+m, \texttt{ID}\rangle}$$

$$\textbf{E-success}: \frac{Ident(P) := Ident(\mathcal{E}.P)}{\langle \mathcal{E}.P, x, \texttt{ID}\rangle \xrightarrow{\mathcal{E}} \langle P, x, \texttt{ID}\rangle} \qquad \frac{n > 0,\ Ident(P) := Ident(\mathcal{E}\nabla(n, P_t, P_e).P)}{\langle \mathcal{E}\nabla(n, P_t, P_e).P, x, \texttt{ID}\rangle \xrightarrow{\mathcal{E}} \langle P, x, \texttt{ID}\rangle}$$

$$\textbf{E-failure}: \frac{n = 0}{\langle \mathcal{E}\nabla(n, P_t, P_e), x, \texttt{ID}\rangle \xrightarrow{\tau} \langle P_t, x, \texttt{ID}\rangle} \qquad \textbf{Exception}: \frac{n > 0, \eth \in \{\mathcal{E}\nabla, \mathcal{A}^{[l,u]}\triangle\}}{\langle \eth(n, P_t, P_e), x, \texttt{ID}\rangle \xrightarrow{\tau} \langle P_e, x, \texttt{ID}\rangle}$$

$$\textbf{E-Delay}: \frac{m \in \mathbb{R}_{\geq 0},\ m < n,\ n' = n-m}{\langle \mathcal{E}\nabla(n, P_t, P_e).P, x, \texttt{ID}\rangle \xrightarrow{m} \langle \mathcal{E}\nabla(n', P_t, P_e).P, x+m, \texttt{ID}\rangle}$$

$$\textbf{E-ChoiceL}: \frac{Ident(P_1) := Ident(E.P_1 + P_2)}{\langle E.P_1 + P_2, x, \texttt{ID}\rangle \xrightarrow{E} \langle P_1, x, \texttt{ID}\rangle} \qquad \textbf{A-ChoiceL}: \frac{m \in \mathbb{R}_{\geq 0},\ Ident(P_1) := Ident(A^{[l,u]} : P_1 + P_2)}{\langle A^{[l,u]} : P_1 + P_2, x, \texttt{ID}\rangle \xrightarrow{A} \langle P_1, x+m, \texttt{ID}\rangle}$$

$$\textbf{E-sync}: \frac{-}{\langle \mathcal{E}.P_1 \| \bar{\mathcal{E}}.P_2, x, \texttt{ID}\rangle \xrightarrow{\tau} \langle P_1 \| P_2, x, \texttt{ID}\rangle} \qquad \textbf{E-AsyncL}: \frac{-}{\langle \mathcal{E}.P_1 \| P_2, x, \texttt{ID}\rangle \xrightarrow{\mathcal{E}} \langle P_1 \| P_2, x, \texttt{ID}\rangle}$$

$$\textbf{A-sync}: \frac{\rho(\mathcal{A}_1) \cap \rho(\mathcal{A}_2) = \emptyset, m = max_t(\mathcal{A}_1, \mathcal{A}_2)}{\langle \mathcal{A}_1 : P_1 \| \mathcal{A}_2 : P_2, x, \texttt{ID}\rangle \xrightarrow{\mathcal{A}_1 \cup \mathcal{A}_2} \langle P_1 \| P_2, x+m, \texttt{ID}\rangle} \qquad \textbf{A-async}: \frac{\rho(\mathcal{A}_1) \cap \rho(\mathcal{A}_2) \neq \emptyset, \neg(\mathcal{A}_1 \prec \mathcal{A}_2)}{\langle \mathcal{A}_1 : P_1 \| \mathcal{A}_2 : P_2, x, \texttt{ID}\rangle \xrightarrow{\mathcal{A}_1} \langle P_1 \| \mathcal{A}_2 : P_2, x+m, \texttt{ID}\rangle}$$

$$\textbf{Event-Action}: \frac{-}{\langle \mathcal{A} : P_1 \| \mathcal{E}.P_2, x, \texttt{ID}\rangle \xrightarrow{\mathcal{E}} \langle \mathcal{A} : P_1 \| P_2, x, \texttt{ID}\rangle} \qquad \textbf{NIL}: \frac{-}{\langle NIL \| P, x, \texttt{ID}\rangle \xrightarrow{\tau} \langle \texttt{Deadlock}, x, \texttt{ID}\rangle}$$

$$\textbf{P-destroy}: \frac{-}{\langle DONE \| P, x, \texttt{ID}\rangle \xrightarrow{\tau} \langle P, x, \texttt{ID} \backslash Ident(DONE)\rangle} \qquad \textbf{P-create}: \frac{P \stackrel{def}{=} P_1 \| P_2, Ident(P_1) := fresh(\textbf{ID}), Ident(P_2) := fresh(\textbf{ID})}{\langle P, x, \texttt{ID}\rangle \xrightarrow{\tau} \langle P_1 \| P_2, x, \texttt{ID} \cup \{Ident(P_1), Ident(P_2)\}\rangle}$$

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 Var, by $\mathcal{C}(Var) = \{x \bowtie v | x \in Var, v \in \mathbb{R}\}$ we define the set of variable constraints of Var where $\bowtie \in \{<, \leq, >, \geq, =\}$.

Definition 4.1 (Parameterized Stopwatch Automaton): A stopwatch automaton over a set of channels Ω is given by a tuple $\langle Q, q^0, Com, C, \mathcal{V}, V^0, \longrightarrow, Act, Inv\rangle$ 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, $\mathcal{V}$ is a set of discrete variables initialized according to the valuation V^0, $\longrightarrow \subseteq Q \times \mathcal{C}(C \cup \mathcal{V}) \times \Omega! \cup \Omega? \cup \{\tau\} \times \Lambda \times Q$ is the transition relation where $\Lambda : \{C \nrightarrow \{0\}\} \cup \{\mathcal{V} \nrightarrow \mathbb{R}\}$ 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 $Inv : Q \rightarrow \mathcal{C}(C)$ associates to each location clock invariants.

We write $q_1 \xrightarrow{g/\lambda/\alpha} q_2$ for a transition $(q_1, g, \lambda, \alpha, q_2) \in \longrightarrow$ outgoing from q_1 to q_2 guarded by g, labeled with λ and performing the action α on clocks and discrete variables. In fact, we have used a long right-arrow ($\longrightarrow$) 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 UPPAAL. 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 **loc** of type Q stating the current location of the process. We also introduce $f \triangleleft g$ as the right overriding function where function g overrides function f for all elements in the intersection of their domains. For all $z \in domain(f \triangleleft g)$,

$$(f \triangleleft g)(z) \triangleq \begin{cases} g(z) & \text{if } z \in domain(g) \\ f(z) & \text{if } z \in domain(f) - domain(g) \end{cases}$$

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 Ω 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 **loc** to the first instance of T_0. The semantics of network $\langle T_0, .., T_n \rangle$ is given by the TTS $\langle \mathcal{S}, \mathcal{S}^0, \rightarrow \rangle$ where $\mathcal{S} = \{s \in val(W) | \forall i \ s \models Inv_i(\mathbf{loc}_i)\}$ is a set of valuations, each one satisfies the local invariants of the involved locations $\mathbf{loc}_i$ of the corresponding active processes. $\mathcal{S}^0 = \bigcup_i \{V_i^0 \cup \{C_i \mapsto \{0\}\}\} \cup \{Id_0.\mathbf{loc} \mapsto q_0^0\}$. $W = \bigcup_i \{Id_i.C\} \cup \bigcup_i \{Id_i.\mathcal{V} \cup \{\mathbf{loc}\}\}$ is the set of discrete and continuous variables of the instances that can be created together with the newly introduced variables **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 instances are valued. The condition $comit(s, Id) = \forall Id' \in s \ com_{Id'}(s(Id'.\mathbf{loc})) \Rightarrow com_{Id}(s(Id.\mathbf{loc}))$ 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_i such that its current location $Id_i.\mathbf{loc}$ corresponds to the current reached location q, and the current system state s satisfies the guard g 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 τ, 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 UPPAAL via statement `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 $Id' := fresh(s)$ thanks to the function $Ident$. The new instance will be immediately initialized through function $fInit$ given by: $fInit(Id', T) = \{Id'.\mathbf{loc} \mapsto T.q^0, \|_{v \in T.\mathcal{V}} \{Id'.v \mapsto T.V^0(v)\}, \|_{c \in T.C} \{Id'.c \mapsto 0\}\}$. 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 `exit()` of rule **Destroy**, stating that the process execution is done, UPPAAL removes such a process from the system.

Rule **Delay** states that a delay d is allowed if the invariants of the current locations ($s(Id.\mathbf{loc})$) of all processes ($Id \in s$) are still satisfied, and no one of the current locations is committed. The time progress function $s \oplus d$ 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 Act. Formally, $s \oplus d = \forall Id \in s, \forall c \in Id.C \ c := s(Id.c) + Act(Id.\mathbf{loc})(c) \times d$.

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

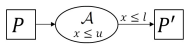
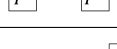
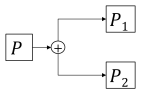
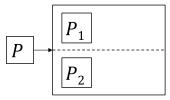
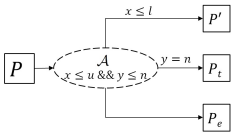
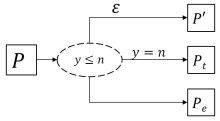
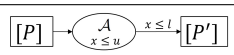
Rule	PACOR	gPACOR
1	$P \stackrel{def}{=} P'$	
2	$P \stackrel{def}{=} \mathcal{A}^{[l,u]} : P'$	
3	$P \stackrel{def}{=} \mathcal{E}.P'$	
4	$P \stackrel{def}{=} P_1 + P_2$	
5	$P \stackrel{def}{=} P_1 \parallel P_2$	
6	$P \stackrel{def}{=} \mathcal{A}^{[l,u]} \triangle (n, P_t, P_e) : P'$	
7	$P \stackrel{def}{=} \mathcal{E} \nabla (n, P_t, P_e).P'$	
8	$[P] \stackrel{def}{=} \mathcal{A}^{[l,u]} : P'$	

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 $\mathcal{A}$, 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

TABLE IV
TRANSITION RELATION OF PSA SEMANTICS

$$\text{Action}: \frac{q \xrightarrow{G/\lambda/a} q' \;\; \exists Id_i \in s | s(Id_i.\mathbf{loc}) = q,\; s \models G,\; comit(s, Id_i)}{s \xrightarrow{\lambda} a(s \lhd \{Id_i.\mathbf{loc} \mapsto q'\})}$$

$$\text{Sync}: \frac{q_i \xrightarrow{G_i/e/a_i} q_i' \;\; q_j \xrightarrow{G_j/\bar{e}/a_j} q_j' \; \exists Id_i \; Id_j \in s | s(Id_i.\mathbf{loc}) = q_i,\; s \models G_i \; s(Id_j.\mathbf{loc}) = q_j,\; s \models G_j,\; comit(s, Id_i) \vee comit(s, Id_j)}{s \xrightarrow{\tau} a_j(a_i(s \lhd \{Id_i.\mathbf{loc} \mapsto q_i', Id_j.\mathbf{loc} \mapsto q_j'\}))}$$

$$\text{Create}: \frac{q \xrightarrow{G/\lambda/spawn\ T()} q' \;\; \exists Id_i \in s | s(Id_i.\mathbf{loc}) = q,\; s \models G,\; comit(s, Id_i),\; Id' := fresh(s)\; Ident([\![T]\!]_s) := Id'}{s \xrightarrow{\lambda} s \parallel fInit(Id', T)}$$

$$\text{Destroy}: \frac{q \xrightarrow{G/\lambda/exit()} q' \; \exists Id_i \in s | s(Id_i.\mathbf{loc}) = q,\; s \models G,\; comit(s, Id_i)}{s \xrightarrow{\lambda} s/Id_i} \quad \text{Delay}: \frac{\forall Id \in s \; s \oplus d \models Inv_{Id}(s(Id.\mathbf{loc})), \neg com_{Id}(s(Id.\mathbf{loc}))}{s \xrightarrow{d} s \oplus d}$$

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 $\oplus$ and arrows leading to possible processes, whereas the parallel composition is represented by a dashed box. The scoped timed action node, with resource requirements $\mathcal{A}$, is shown in Rule 6. It has 3 outgoing transitions leading to a normal process (P'), a time-out handler (P_t), and an exception handler (P_e), respectively. 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 $\mathcal{E}$ 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.

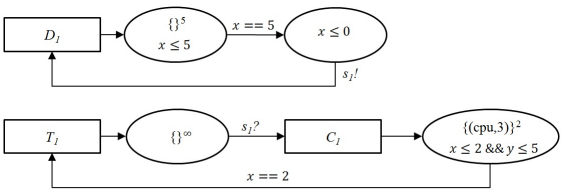


Fig. 2. gPACoR model of Example 3.1

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, $\mathcal{A}_1, \mathcal{A}_2, ...$ for resource requirements, and $A_1, A_2, ...$ 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.
- $Request(\mathcal{A})$ is a resource request according to the resource requirement $\mathcal{A}$,
- $Release(2^R)$ is a function that releases a set of resources (2^R),
- $Available(2^R)$ is a function stating whether a set of resources (2^R) is available,
- $LockPreemp(pid)$ 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 $Request(\mathcal{A})$ according to a resource requirement $\mathcal{A}$ before entering the timed action location. The PSA process should release resources, using the function $Release(2^R)$, 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 $Available(2^R)$. If a process wants an exclusive (non-preemptive) use of resources, it sets itself to "non-preemptive" using $LockPreemp(pid)$.

TABLE VI
TRANSLATION RULE 2-5

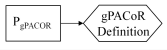
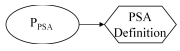
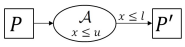
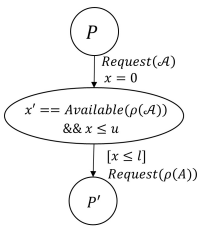
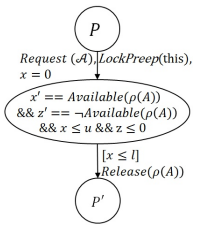
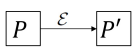
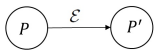
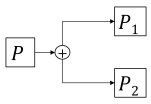
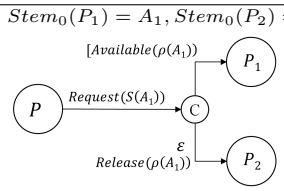
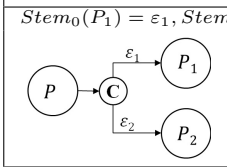
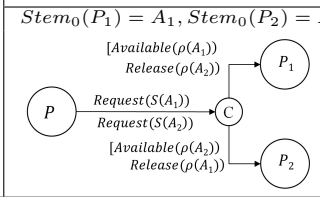
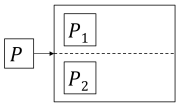
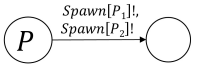
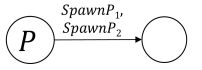
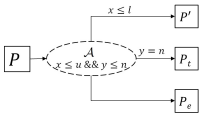
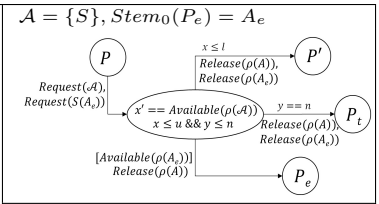
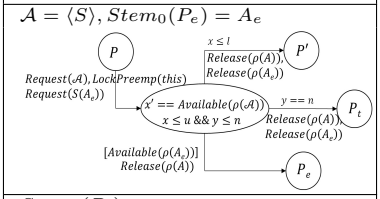
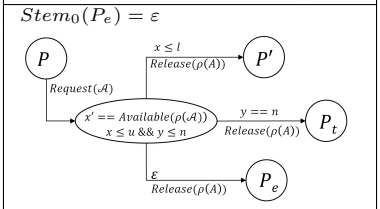
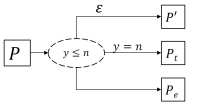
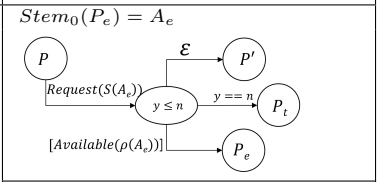
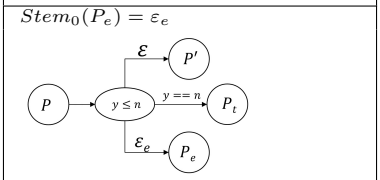
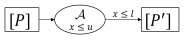
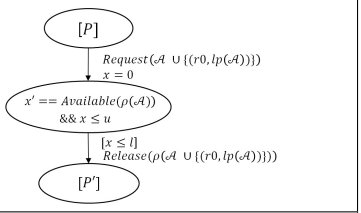
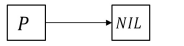
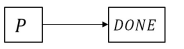
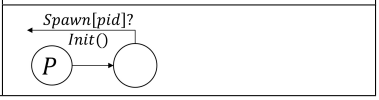
Rule	gPACoR	PSA
0		
2-1		
2-2		
3		
4-1		
4-2		
4-3		
5-1		
5-2		

TABLE VII
TRANSLATION RULE 6-10

Rule	gPACoR	PSA
6-1		
6-2		
6-3		
7-1		
7-2		
8		
9		
10-1		
10-2		

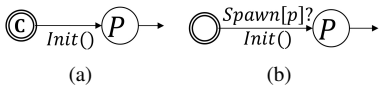
Rule 3 shows how gPACoR events are translated to a set of transitions into the corresponding PSA model. As shown in Rule 4, the Choice operator $\oplus$ is replaced by the commit connector ⓒ of UPPAAL. The Choice operator has alternatives of possible actions. In case of timed actions, the availability 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. ($Request(S(A_1))$), before making a choice and checked to be usable (transitions are enabled) when making the choice, e.g. $[Available(\rho(A_1))]$. 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, $Spawn[pid]!$, whereas the dynamic instantiation is supported by the UPPAAL statement $Spawn$ 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 $LockPreemp(pid)$. 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 ε_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 $(r, lp(\mathcal{A}))$ 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 $exit()$ 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 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), excepts 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 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 C1 and C2 execute the function request() to request cpu, and the outgoing transitions from them call the function release() to release the resource.



(a) (b)

Fig. 3. Initial locations of PSA

Theorem 4.1: The TTS semantics restricted to τ 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 removed 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.

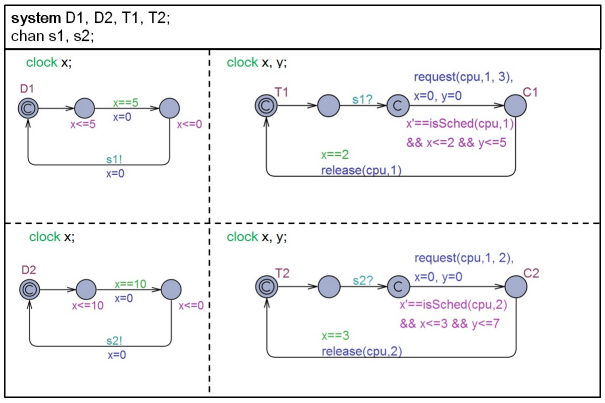


Fig. 4. PSA Model for Example 3.1

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.

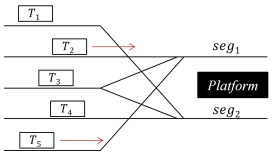


Fig. 5. Platform system

A. Analysis using UPPAAL and UPPAAL SMC

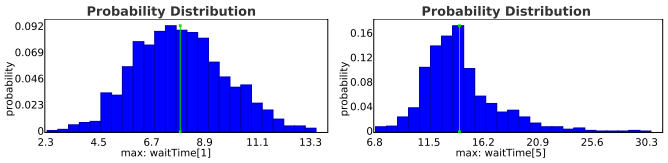
The analysis of the train platform system has been done using two verification methods, symbolic and statistical model checking. We use UPPAAL to check with symbolic model checking that there is no deadlock in the system. With UPPAAL SMC we generate the probability distribution for the waiting time of each train.

TABLE VIII
TRAIN PLATFORM SYSTEM IN PACOR

$$
\begin{aligned}
System &\stackrel{def}{=} [\ TrnReqP_{1,pri_1} \parallel TrnReqP_{2,pri_2} \parallel ... \parallel TrnReqP_{T,pri_T}\ ]_{\{seg_j\}} \qquad 1 \le j \le Seg \\
TrnReqP_{i,k} &\stackrel{def}{=} \emptyset^{\infty} \triangle (wt, TrnReqP_{i,k+1}, \sum_{j=1}^{Seg} \langle (seg_j, k) \rangle^{[0,st]} : TrnLeaveP_{j,i}) : NIL \\
TrnLeaveP_{j,i} &\stackrel{def}{=} \emptyset^{mt} : TrnArvl_i \\
TrnArvl_i &\stackrel{def}{=} \emptyset^{rt} : TrnArvl_i + TrnReqP_{i,k}
\end{aligned}
$$

TABLE IX
THE MAXIMUM WAITING TIME WITH $st = 10 \pm 5$ AND $wt = 10$

	Case 1 ($Seg = 2$)		Case 2 ($Seg = 2$)		Case 3 ($Seg = 3$)	
i	pri_i	Wait Time	pri_i	Wait Time	pri_i	Wait Time
1	1	14.14±0.22	5	8.04±0.12	5	4.13±0.13
2	1	14.22±0.22	4	9.42±0.14	4	4.40±0.15
3	1	14.16±0.22	3	11.76±0.16	3	4.61±0.15
4	1	14.34±0.23	2	14.41±0.19	2	7.67±0.10
5	1	14.20±0.22	1	20.94±0.31	1	9.15±0.08



(a) Track 1 (b) Track 5

Fig. 6. Probability distribution of the maximum waiting times of Track 1 with priority 5 and Track 5 with priority 1 in Case 2

In order to check the safety of the system, first, we use the following query:

A[ ] not deadlock

which inquiries whether there is a deadlock in the system.

In addition to the analysis of the safety, UPPAAL SMC enables us to analyze the response-time of the system. We use the following query of UPPAAL SMC to inquire what is the average of the maximum waiting time of the train for the assignment of the segment:

E[<=simTime;simCount](max:waitTime[i])

This query generates the average of the maximum waiting time of the train for a given simulation time ($simTime$) and number of simulations ($simCount$). Table IX shows the variability of the maximum waiting time of trains according to different sets of priorities of the tracks: if the same priority is given to all the tracks, the average of the maximum waiting time for trains are almost equal. However, for different priorities, the trains running on tracks of higher priorities have shorter waiting time than the ones running on tracks with lower priorities. Fig. 6 shows the probability distributions of the maximum waiting time, which follow a Gaussian distribution.

VI. CONCLUSION

In this paper we have defined PACOR which is an updated and improved derivative of the classical process algebra ACSR. The formal semantics of PACOR is given in the form of a timed transition system. We have provided translation rules from PACOR to PSA which preserves the semantics. The translation is not implemented yet, but it is designed to utilize new features of UPPAAL, 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 UPPAAL and UPPAAL SMC.

REFERENCES

[1] J. P. Bodeveix, A. Boudjadar, and M. Filali. An alternative definition for timed automata composition. In *ATVA'11*, pages 105–119. LNCS 6996, 2011.

[2] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikučionis, U. Nyman, and A. Skou. Hierarchical scheduling framework based on compositional analysis using uppaal. In *Proc of FACS 2013. To appear*.

[3] A. Boudjadar, F. Vaandrager, J.-P. Bodeveix, and M. Filali. Extending uppaal for the modeling and verification of dynamic real-time systems. In *FSEN 2013*, LNCS, pages 111–132, 2013.

[4] P. Brmond-Grgoire and I. Lee. A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189(12):179 – 219, 1997.

[5] F. Cassez and K. Larsen. The impressive power of stopwatches. In *Proc. of CONCUR 2000*, pages 138–152. Springer, 2000.

[6] D. Clarke, H. Ben-Abdallah, I. Lee, H. liang Xie, and O. Sokolsky. Xversa: An integrated graphical and textual toolset for the specification and analysis of resource-bound real-time sytems. In *CAV*, volume 1102 of *LNCS*, pages 402–405. Springer, 1996.

[7] D. Clarke, I. Lee, and H. liang Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3, 1995.

[8] CRAFTERS. project website. http://www.crafters-project.org/.

[9] A. David, K. G. Larsen, A. Legay, and M. Mikucionis. Schedulability of herschel-planck revisited using statistical model checking. In *ISoLA (2)*, volume 7610 of *LNCS*, pages 293–307. Springer, 2012.

[10] A. David, K. G. Larsen, A. Legay, and D. B. Poulsen. Statistical model checking of dynamic networks of stochastic hybrid automata. In *Proceedings of AVoCS13*, volume to appear, page to appear, 2013.

[11] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. V. Weerdenburg. The formal specification language mcrl2. In *In Proceedings of Methods for Modelling Software Systems*. MIT Press, 2007.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[13] A. S. Jeffrey, S. A. Schneider, and F. W. Vaandrager. A comparison of additivity axioms in timed transition systems. Technical report, Amsterdam, The Netherlands, The Netherlands, 1993.

[14] I. Lee, P. Bremond-Gregoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, 82(1):158–171, 1994.

[15] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[16] A. Philippou, I. Lee, and O. Sokolsky. Pads: An approach to modeling resource demand and supply for the formal analysis of hierarchical scheduling. *Theor. Comput. Sci.*, 413(1):2–20, 2012.

[17] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Theor. Comput. Sci*, pages 314–323, 1988.

APPENDIX A
REALIZATION OF RESOURCE OPERATIONS IN UPPAAL

In order to realize the priority relation in UPPAAL, we implemented the prioritizing functions and resource availability functions using the user-defined functions of UPPAAL.

Listing 1. Data structures for PACoR process and resources

```
typedef struct  {
    pid_t     pid;
    ppri_t    pri ;
    bool      preemptive;
} proc_t;

typedef struct  {
    pid_ti    len;
    pid_ti    elem[pid_n];
} rq_t;

rq_t      rq[ rid_t ]  ;
proc_t    proc[pid_t];
```

Listing 1 shows data structures for processes and resources. In order to prioritize processes sharing resources, each process is given a process identity (pid), a priority (pri), and the preemptiveness (preemptive). The process can have different priorities for each resource that it requires. A resource in PACoR is realized by a queue, which consists of a list of priority-ordered process identities and the queue length. For a given resource. the resource queue keeps the order of processes according to priorities that each process presents to use it.

Listing 2. Prioritizing functions

```
void insert_at(rid_t  rid ,  pid_t  pid,  pid_t  place,  pid_t  pid){
    pid_ti  i  = rq[ rid ]. len  ;
    for(i  = rq[ rid ]. len;  i  > place; i--){
        rq[ rid ]. elem[i]  = rq[ rid ]. elem[i-1];
    }
    rq[ rid ]. elem[place] = pid;
    rq[ rid ]. len++;
}

void request(rid_t  rid ,  pid_t  pid,  ppri_t  pri ){
    pid_ti  place=0;
    test [ pid]  = pri ;
    proc[pid]. pri  = pri ;
    if  (rq[ rid ]. len  > 0) place =
            (proc[rq[ rid ]. elem[0]]. preemptive? 0:1);
    while(place < rq[rid]. len  &&
    proc[rq[ rid ]. elem[place]]. pri  > proc[pid]. pri ){
        place++;
    }
    insert_at(rid ,  place,  pid);
}

void release(rid_t  rid ,  pid_t  pid){
    pid_ti  i =0;
    while(i  < rq[rid ]. len  && rq[rid ]. elem[i]!= pid)  i ++;
    while(i  < rq[rid ]. len-1){
        rq[ rid ]. elem[i]  = rq[ rid ]. elem[i+1];  i ++;
    }
    rq[ rid ]. elem[i]  = 0;
    rq[ rid ]. len--;
}
```

Listing 2 shows the realization of resource request and release of PACoR processes. The preemptiveness for a resource depends on the preemptiveness of the current resource owner process specified in the variable preemptive of a process. If the variable preemptive of a resource owner process is true, a process with higher priority can take away the resource from the owner process. Otherwise, a newly resource-requesting process is placed at the second position in the queue even if it has the highest priority in the queue.

The function request() calculates the order of a process in the present resource queue status according to its priority. The function insert_at() inserts a new process into the queue. The function release() deletes a process from a resource queue. It searches and removes a given process id in resource queue, deletes it, and re-orders all the elements of processes.

Listing 3. Resource availability function

```
int [0,1]  isSched(rid_t  rid ,  pid_t  pid)
                {return (rq[ rid ]. elem[0] == pid)? 1:0;}
int [0,1]  isNotSched(rid_t rid ,  pid_t  pid)
                {return (rq[ rid ]. elem[0] == pid)? 0:1;}
```

The functions isSched(rid,pri) and isNotSched(rid,pri are used to check if a resource represented by a resource queue (rid) is available to a process pid. The function isSched() returns 1 when pid is the first element in the resource queue of rid, and otherwise, it returns 0. The function isNotSched() is opposite to isSched(), returning 1 if the process is not scheduled.

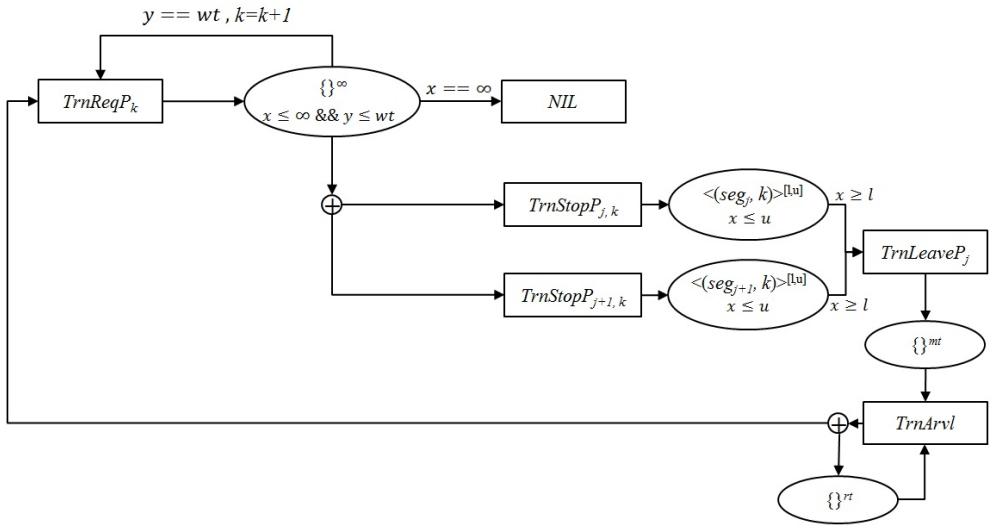
APPENDIX B
PSA MODEL FOR TRAIN PLATFORM SYSTEMS

Fig. 7 shows the gPACoR and the corresponding PSA model of the train platform systems translated from the PACoR model given in Section . As shown Fig. 7, the PSA model has almost same structure as the gPACoR model, except for NIL process and the Choice connector. The NIL process has the incoming transition which is never taken because the clock x cannot ever hold. The Choice connector is removed since all alternative transitions have the same possibility to be taken as the timed action execution, thus they directly linked to the time action location. All the same name of gPACoR can be found in the PSA model. The clock variable waitTime[pid] is added to the PSA model after the translation in order to analyze the waiting time of a train before entering a platform. The urgent event dummy is also added in order for transitions depending on conditions to be taken urgently.

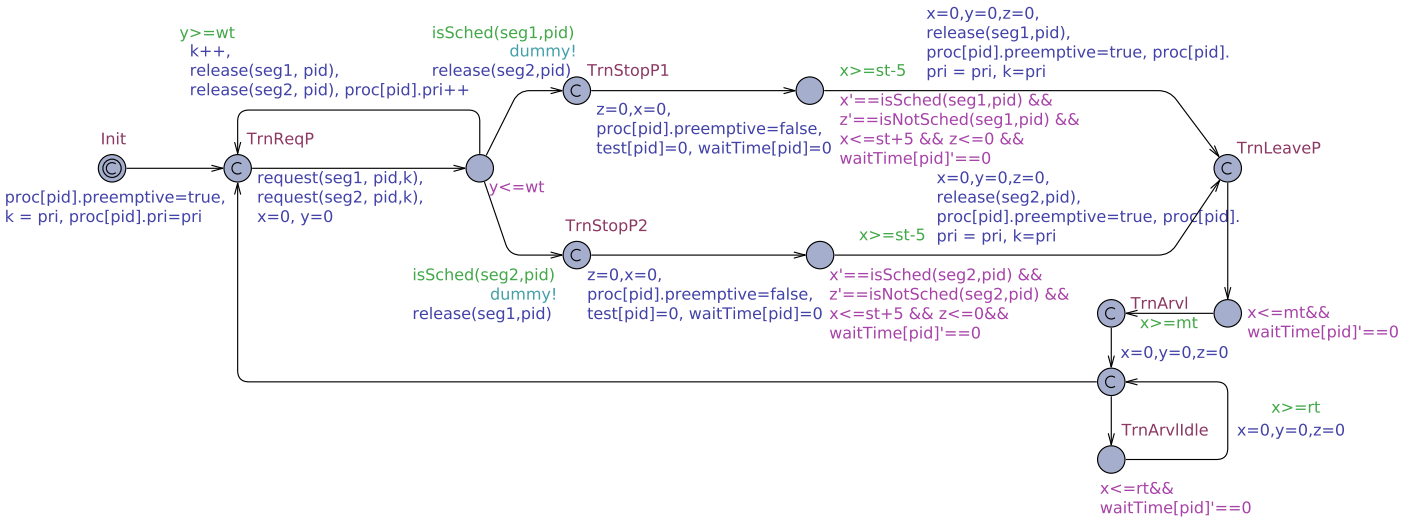
APPENDIX C
PROOF OF THEOREM 4.1

The proof consists in showing that each TTS semantics (direct, translation-based) of PACoR simulates the other. Because of space limitation, we only focus on one simulation, from direct semantics to the translation-based one, of the transition relations where we provide the minimum elements making our theorem clearly satisfied. In fact, the proof relies on the fact that:

- The semantics of each PACoR event is represented by either an event transition (rule **E-success**) or a delay (rule Delay) followed by the event transition if the PACoR event is not urgent. Similarly, in the translation from PACoR to PSA, each event corresponds to en edge performing such an event (Rule 3) where the PSA template may have a wait if the event is not urgent. Such a waiting time corresponds to a delay transition (rule Delay), in the TTS semantics of PSA, depending on the PSA location invariant that depends in turn to the urgentness of the PACoR event. In both cases, the target states of the



(a) gPACoR model



(b) PSA model

Fig. 7. gPACoR and PSA models of the train platform system

event-transitions match each other in terms of simulation relation.

- A PACoR timed action semantics corresponds to a TTS transition performing such an action (rule **A-success**) when resources are available. Such a PACoR transition corresponds to a sequence of 2 PSA edges t_1 and t_2 (Rule 2-1, 2-2, 6-1, 6-2, 6-3). The first edge t_1 requests resources whereas the second edge t_2 performs such an action and release resources. In fact, the use of resources is implemented by a stay at the source location of t_2. One can remark that the TTS state corresponding to the source location of t_2 matches perfectly with the (direct semantics) TTS states of the event transition in terms of the simulation relation. Such an extra internal transition, corresponding to the PSA edge t_1 for resource request, can be ignored according to the weak bisimulation relation we considered. Moreover, the target state of the action transition (direct semantics)matches with the target state of the semantics of edge t_2. Thus, our simulation relation is satisfied in the case of timed action transitions.

The proof of the simulation relation from translation-based semantics and direct semantics is similar to the proof sketch given above.