Process-Algebraic Analysis of Timing and Schedulability Properties

Anna Philippou Oleg Sokolsky

January 4, 2007

Abstract

In this chapter, we present an overview of how timing information can be embedded in process-algebraic frameworks. We concentrate on the case of discrete-time modeling. We begin by discussing design approaches that have been adopted in different formalisms to model time and time passage, and how the resulting mechanisms interact with one another and with standard untimed process-algebraic operators. We proceed to give an overview of ACSR, a timed process algebra developed for modeling and reasoning about timed, resource-constrained systems. In doing this, ACSR adopts the notion of a *resource* as a first-class entity, and it replaces maximal progress, employed by other timed process algebras, by the notion of *resource-constrained* progress. ACSR associates resource-usage with time passage, and implements appropriate semantic rules to ensure that progress in the system is enforced as far as possible while simultaneous usage of a resource by distinct processes is excluded. In addition, ACSR employs the notion of priorities to arbitrate access to resources by competing processes. Finally, we illustrate the use of ACSR for the schedulability analysis of a realistic real-time system problem.

1 Introduction

Process algebras, such as CCS [13], CSP [9], ACP [3] and others, are a well-established class of modeling and analysis formalisms for concurrent systems. They can be considered as high-level description languages consisting of a number of operators for building processes including constructs for defining recursive behaviors. They are accompanied by semantic theories which give precise meaning to processes, translating

each process into a mathematical object on which rigorous analysis can be performed. In addition, they are associated with axiom systems which prescribe the relations between the various constructs and can be used for reasoning about process behavior. During the last two decades, they have been extensively studied and proved quite successful in the modeling and reasoning about various aspects of system correctness.

In a process algebra, there exist a number of elementary processes as well as operators for composing processes into more complex ones. A system is then modeled as a term in this algebra. The rules in the operational semantics of an algebra define the steps that a process may take in terms of the steps of the subprocesses. When a process takes a step, it evolves into another process. The set of processes that a given process P can evolve into by performing a sequence of steps defines the state space of P. The state space is represented as a labeled transition system (LTS) with steps serving as labels. The notation for a process P performing a step a and evolving into a process P' is $P \xrightarrow{a} P'$.

State space exploration techniques allow us to analyze properties of processes, for example, identify deadlocks that the system may have.

1.1 Conventional process algebras

Instead of giving a formal definition for some process algebra, we will begin by supplying intuition for the design of a process-algebraic formalism. Without being specific to any particular process algebra, we will introduce basic concepts that are present in a typical process algebra and highlight the relationship between them. This exposition will help us illustrate how the introduction of timing into a process algebra, discussed in later sections, affects the design of a formalism.

Syntax and semantics. The syntax of a process algebra is defined by a collection of operators that give rise to a set of processes. Each operator has a non-negative arity. Operators of arity 0 represent elementary processes. Operators of arity n take n processes and form a new composite process from them. A semantic definition for such an algebra is formalized by means of structural operational semantics: for each operator a set of rules is supplied that prescribes how steps of the constituent processes are transformed into a step of a composite process. Examples of such rules are given later in the section.

Basic execution steps. Basic execution steps of a process P are taken from a set called the *alphabet* of P, which we denote as $\mathcal{A}(P)$. Since execution steps serve as transition labels in the LTS of P, we generically refer to the elements of the alphabet as labels. Labels in the alphabet of conventional process algebras are often called *events* or *actions*. Here, we will refer to them as *events*, reserving the term *actions* to a special kind of steps which will be introduced in Section 3. Real-time extensions of process algebras also introduce additional kinds of labels to account for time, as discussed in Section 2. The nature of events depends on the modeling domain: it can correspond, for example, to a method call in the model of a software system or to receiving a packet from the network in the model of a communication protocol.

Common operators and rules. The syntax of a typical process algebra is given by the following BNF definition

$$P \quad ::= \quad \text{NIL} \mid e.P \mid P + P \mid P \parallel P \mid P \backslash A \mid C$$

where e and C range over a set $\mathcal{A}$ of events and a set $\mathcal{C}$ of process constants associated with the process algebra, respectively, and $A \subset \mathcal{A}$. Each operator is given precise meaning via a set of rules which, given a process P, prescribe the possible transitions of P, where a transition of P has the form $P \xrightarrow{e} P'$, specifying that process P can perform event e and evolve into process P'. The rules themselves have the form

$$\frac{T_1, \ldots, T_n}{T} \quad \phi$$

which is interpreted as follows: if transitions $T_1, \ldots, T_n$, can be derived, and condition ϕ holds, then we may conclude transition T. Conditions $T_1, \ldots, T_n$, describe possible transitions of the components of a system and may be absent for some rules. We proceed to discuss each of the above operators and ways in which they are interpreted in different process-algebraic formalisms.

- NIL. The *inactive process*, represented by the operator NIL, does not have any rules associated with it and thus cannot perform any steps.

- $e.P$. The most basic operator of a process algebra is the *prefix* operator. If P is a process then $e.P$ is also a process, for every event e. This process performs event e and then behaves as P. The rule associated with a prefix operator usually has the form

$$(\mathsf{ACT}) \qquad \frac{-}{e.P \xrightarrow{e} P}$$

An alternative to the prefix operator is a more general *sequential composition* operator $P_1 \cdot P_2$ for arbitrary processes P_1 and P_2. We refer the reader to [3] for details.

- $P_1 + P_2$. Another commonly-used operator is the choice operator, $P_1 + P_2$, which yields a process that spontaneously evolves into P_1 or P_2. The rules for the choice operator are

$$(\mathsf{SUM}) \qquad \frac{P_1 \xrightarrow{e} P_1'}{P_1 + P_2 \xrightarrow{e} P_1'},$$

and a symmetric rule for P_2. We say that the first event performed by P_1 or P_2 resolves the choice.

- $P_1 \,\|\, P_2$. One of the main points of distinction between different process algebras is the *parallel composition* operator, which we denote here as $P_1 \,\|\, P_2$. Details of the definition for this operator specify whether processes in the model execute synchronously or asynchronously and describe the means of communication and synchronization between concurrent processes. In the case of asynchronous execution, processes can take steps independently. This case is expressed by the rule

$$(\mathsf{PAR_A}) \qquad \frac{P_1 \xrightarrow{e} P_1'}{P_1 \,\|\, P_2 \xrightarrow{e} P_1' \,\|\, P_2}$$

Note that here, unlike the choice operator, P_1' remains in the scope of the parallel operator. P_2 does not make a step in this case. There is a symmetric rule for P_2.

In the case of a synchronous execution, all concurrent processes are required to take a step. A generic rule for the synchronous case can be expressed as

$$(\mathsf{PAR_S}) \qquad \frac{P_1 \xrightarrow{e_1} P_1' \quad P_2 \xrightarrow{e_2} P_2'}{P_1 \,\|\, P_2 \xrightarrow{e} P_1' \,\|\, P_2'}$$

The relationship between e and e_1, e_2 captures the synchronization details. In what is known as *CCS-style* synchronization, the alphabet $\mathcal{A}$ is partitioned into the sets of input events, denoted as $e?$, output events, denoted as $e!$, and a distinguished internal event τ. A handshake synchronization between two processes turns an input event and a matching output event into an internal event, represented by the rule

$$(\mathsf{PAR_{CCS}}) \qquad \frac{P_1 \xrightarrow{e?} P_1' \quad P_2 \xrightarrow{e!} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'}$$

In *CSP-style* synchronization, components of a process P must synchronize on a predefined set $A \subseteq \mathcal{A}$ and we have that $e = e_1 = e_2 \in A$. In other formalisms, the result of a synchronization is a composite event, for example, $e_1 \cdot e_2$. The general case is called the *ACP-style* synchronization, which employs a *communication function* f. The communication function is a partial function that maps pairs of events to events. The rule

$$(\mathsf{PAR_{ACP}}) \qquad \frac{P_1 \xrightarrow{e_1} P_1' \quad P_2 \xrightarrow{e_2} P_2'}{P_1 \parallel P_2 \xrightarrow{f(e_1,e_2)} P_1' \parallel P_2'} \quad f(e_1, e_2) \neq \perp$$

is equipped with a side condition that makes the rule applicable only if the communication function is defined for e_1 and e_2.

- $P\backslash A$. This process represents the *hiding* or *restriction* operator representing the process P when events from set A are hidden. This is another operator interpreted differently in distinct process algebras. These interpretations are closely related to the communication style adopted. In particular, in the ACP style of communication, hiding of an action prohibits the action from taking place altogether. On the other hand, in CCS, where the set of actions contains input and output actions along channels, hiding of a channel e within a system P prevents actions $e?$ and $e!$ from taking place on the interface of P with the environment, but not within the components of P.

- C. A common mechanism for providing recursion in process algebra is *process constants*. A set of process constants, $\mathcal{C}$, is assumed. Each $C \in \mathcal{C}$ is associated with a definition $C \stackrel{\text{def}}{=} P$, where the process P may contain occurrences of C, as well as other constants. The semantics of process constants is given by the following rule:

$$(\mathsf{CON}) \qquad \frac{P \xrightarrow{e} P'}{C \xrightarrow{e} P'} \quad C \stackrel{\text{def}}{=} P$$

Equational Theory. A significant component of a process-algebraic framework is its equational theory, a set of *axioms* or *equations* that equate process expressions of the language. Some process algebras, such as [3] and its derivatives, are even defined equationally since their designers consider this to be the most intuitive

starting point for understanding the algebra and its operators. In the case that an algebra's semantics are given operationally, as discussed above, this theory is produced on the basis of an *equivalence relation*, which deems two processes to be related if and only if their respective LTS's are "similar", according to a specific notion of similarity. Based on such an equivalence relation, an equational theory is *sound* if all of its axioms equate equivalent processes and, it is *complete*, if any pair of equivalent processes can be shown to be equal by the equational theory.

The importance of equational theories is twofold. On one hand, they shed light on the subtleties of a language's operators and the interactions that exist between them. As such, they can be useful for revealing differences between different algebras their operators and their associated equivalences. On the other hand, they support system verification by equational reasoning. Note that, in order to support *compositional* reasoning, they are typically defined over *congruence* relations, that is, relations which are preserved by all of a language's operators. One of the most common such relations is *strong bisimulation* [13, 17]. Strong bisimulation is an equivalence relation which respects the branching structure of processes. Intuitively two processes are bisimilar if they can simulate each other's behavior step by step. In Table 1.1 we present a subset of a basic process algebra's equational theory.

Table 1.1: **Axioms of a typical process algebra**

(Sum1)	$P + \text{NIL} = P$	(Par1)	$P \parallel \text{NIL} = P$
(Sum2)	$P + Q = Q + P$	(Par2)	$P \parallel Q = Q \parallel P$
(Sum3)	$(P + Q) + R = P + (Q + R)$	(Par3)	$(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$
(Hide1)	$\text{NIL}\backslash A = \text{NIL}$	(Hide2)	$(a.P)\backslash A = a.(P\backslash A)$, if $a \notin A$

We note that the inactive process serves as the *unit process* for the parallel composition and choice operators (Axioms (Sum1), and (Par1)).

An important axiom in the equational theory of most conventional process algebras is the *expansion theorem*, which allows to rewrite a process containing the parallel composition operator to an equivalent process without parallel composition. The expansion theorem allows simpler analysis algorithms that operate

on the process LTS. Application of the expansion theorem, in CCS, results in the following equation

$$e?.P \parallel e!.Q = e?.(P \parallel e!.Q) + e!.(e?.P \parallel Q) + \tau.(P \parallel Q)$$

whereas, in CSP, it yields

$$e.P \parallel e.Q = e.(P \parallel Q).$$

It is interesting to note that the interplay between the parallel composition and the restriction operator gives rise to $(e?.P \parallel e!.Q)\backslash\{e\} = \tau.(P \parallel Q)$ in CCS and $(e.P \parallel e.Q)\backslash\{e\} = \text{NIL}$ in CSP. This highlights the fact that, while in CCS hiding of a channel within a system does not preclude a handshake communication between components of the system, in CSP the action is prevented from arising altogether.

2 Modeling of Time-Sensitive Systems

Conventional process algebras offer a way of modeling and analyzing qualitative, functional aspects of system behavior. Several *real-time process algebras* extend conventional process algebras to add the quantitative timing aspect to the system model. In this section, we look at the possible ways to add timing information to a process-algebraic model and discuss the effect of such extensions on the rules of the algebra and the interactions between them. Classification ideas may be found in [15].

Two-phase execution. The approach taken by most real-time process algebras is to introduce a new kind of labels into the process alphabet to represent the progress of time. Labels of this kind can either be real or rational numbers, or new symbols that are not present in the alphabet of the untimed process algebra. The former approach is known as a *continuous time* extension, and a step labeled by a number represents advancing the time by the specified amount. The latter approach is called *discrete time*, and a step labeled by a label of the new kind represents time advancing by a fixed amount. We refer to such steps as *ticks*. All other steps of a process are assumed to be instantaneous.

Execution of a process proceeds in two phases. Initially, a sequence of instantaneous steps are executed at the same time instance, followed by a time step that advances time. An execution that contains an infinite sequence of instantaneous steps without allowing a time step to take place is called *Zeno*. Clearly, such

executions do not have a physical meaning and need to be ruled out either by the semantics of the formalism or by means of analysis.

Progress. Conversely, a process can choose to pass time indefinitely even though it is capable of performing instantaneous steps. While not as harmful as Zeno behaviors, such idling behaviors are also undesirable and should be avoided. Therefore, most formalisms include some means of ensuring progress in an execution. Many real-time process algebras employ the notion of *maximal progress*, which forces synchronization between concurrent processes, whenever it can occur, to occur before the next time step. Some process algebras, such as CSA [7] and ACSR [11], employ a more restricted notion of progress, as described below.

Global vs. local time. Most real-time process algebras employ an implicit global clock that is shared by all processes in the system. This means that time progress is *synchronous*, that is, time can advance only if all processes in the system agree to pass time together. This is in contrast to the execution of instantaneous steps of different processes, which may be interleaved even though they are logically simultaneous.

The assumption of uniform global time may be restrictive when modeling distributed systems, where each node may have its own clock that advances at its own rate. To address this problem, CSA [7] considers a multi-clock process algebra. In this algebra, steps can have different time progress labels that correspond to a tick of a particular clock, and only the processes that are within the scope of that clock are required to participate in that step.

2.1 Discrete-time Modeling

In this chapter, we concentrate on discrete-time modeling. For details on continuous-time process algebras, the interested reader is referred to [1, 25, 21]. We consider real-time process algebras that assume a global clock, shared by all concurrent processes, which advances in discrete increments. A tick of that clock is represented by a special label ϵ, which is added to the process alphabet. Events inherited from the untimed versions are considered to be instantaneous, thus are kept independent from the passage of time. Process algebras that fall into this category are ATP [16], TPL [8], TCCS [14] and several ACP flavors [2, 1]. The process algebra ACSR, discussed in Section 3, also assumes a global clock but has a richer structure for time

progress labels.

Syntactic extensions. One of the first steps in designing a real-time process algebra is to decide whether time progress steps are included into the syntax of the process terms. In a conventional process algebra, every event e can appear in the event-prefix operator $e.P$. TCCS and TPL treat time steps in the same way by introducing a *time prefix* operator $\epsilon.P$, with the obvious rule:

$$(\mathsf{TACT}) \qquad \frac{-}{\epsilon.P \xrightarrow{\epsilon} P}$$

Other process algebras, such as ATP, do not have time prefix operators and extend the semantics for other operators with rules about time steps.

Extending operators with rules for time progress. Time progress rules can be added to any operator in the algebra. However, decisions to such rules for different operators are not independent and allow the algebra designers to offer interesting modeling devices. Below, we consider different design choices and look at the interplay between them.

Conventional process algebras usually have one kind of inactive process, which we have denoted by NIL. In real-time process algebras, it is useful to distinguish between a completed process, which we will call δ, and a deadlocked process, which we will call NIL. The semantic difference is, of course, that the completed process cannot perform any actions but allows time progress. That is, it is equipped with the rule

$$(\mathsf{IDLE}) \qquad \frac{-}{\delta \xrightarrow{\epsilon} \delta.}$$

On the other hand, the deadlocked process does not have any rules associated with it, thus preventing time from progressing any further. Process algebras featuring this time-blocking deadlocked process include TCCS, ACSR and ACP_ρ [2]. By this process it is possible to model abnormal conditions that require attention, and can be employed for detecting anomalies in system models (for example, in ACSR, the presence of timelocks can be used to test for schedulability of a system model).

It is worth noting here that if the algebra has the time-prefix operator, then the completed process can be represented by a secondary operator, defined by the recursive definition $\delta \stackrel{\text{def}}{=} \epsilon.\delta$.

Next, we consider event prefix. Here, we may distinguish two different approaches on the treatment of event execution.

- A first approach, adopted by the process algebra TPL, allows events to be so-called *lazy* or *patient* with respect to time passage. To implement this, the prefix operator is accompanied by the rule

  (WAIT) $$\frac{-}{a.P \xrightarrow{\epsilon} a.P.}$$

  signifying that the process $a.P$ can idle indefinitely. The motivation behind this law is the intention for all processes to be patient until the communications in which they can participate become enabled. Then, progress in the system is assured by appropriate treatment of the parallel composition operator as we discuss below. Note that, if events are lazy, there is no need for the time-prefix operator.

- Alternatively, the algebra designed has an option of making events *urgent*. In this case, the event-prefix operator cannot idle and progress is ensured. This approach is adopted in ATP and TCCS. In the case of TCCS, time passage is enabled (solely) by the aid of a time-prefix operator, whereas ATP features a unit delay operator. In some process algebras, such as TPL, urgency of events only applies to the internal action.

  However, modelers using a process algebra with urgent events can inadvertently block time progress completely, introducing unwanted Zeno behavior in the model. In ATP, the term *well-timed* systems has been coined up to refer to systems that contain no Zeno behavior. In general, there is no syntactic characterization of well-timed systems. It is, however, possible to discover Zeno behavior by state space exploration.

Continuing with the choice construct, we may see that the majority of timed process algebras proposed in the literature implement *time determinism*. That is, they define the choice construct so that it postpones the resolution of the choice between its summands in the case of system idling. This is described by the following rule:

(TSUM) $$\frac{P_1 \xrightarrow{\epsilon} P_1', P_2 \xrightarrow{\epsilon} P_2'}{P_1 + P_2 \xrightarrow{\epsilon} P_1' + P_2'}.$$

Note, however, that in an algebra with urgent events a process may not be able to pass time and the choice may have to be resolved before time advances.

As far as parallel composition is concerned, we first observe that time progresses synchronously in all parallel components, that is, timed actions can be performed only if all the system components agree to do so:

$$(\mathsf{TPAR}) \qquad \frac{P_1 \xrightarrow{\epsilon} P_1', P_2 \xrightarrow{\epsilon} P_2'}{P_1 \parallel P_2 \xrightarrow{\epsilon} P_1' \parallel P_2'}$$

Specifically, in the presence of a communication function f, we have that $f(e_1, e_2) = \perp$ if one of e_1, e_2 is ϵ but not both.

Further details concerning the interaction of time passage and the parallel composition construct depend on the synchronization style featured in an algebra as well as the nature of events. In the case of CCS-style communication and patient events, we may see maximal progress implemented by the following rule:

$$(\mathsf{TPAR}_{\mathsf{MP}}) \qquad \frac{P_1 \xrightarrow{\epsilon} P_1', P_2 \xrightarrow{\epsilon} P_2', P_1 \parallel P_2 \not\xrightarrow{\tau}}{P_1 \parallel P_2 \xrightarrow{\epsilon} P_1' \parallel P_2'}$$

This specifies that $P \parallel Q$ may delay if both components may delay and no internal communication between the two is possible. It can be encountered in TPL, where, for example, in the presence of patient actions,

$$a!.P + b?.Q \xrightarrow{\epsilon} a!.P + b?.Q$$

allows time to pass, whereas, the composition $(a!.P + b?.Q) \parallel a?.\mathrm{NIL}$, does not. We should also note that under this rule, the parallel operator construct does not preserve Zenoness: $P \stackrel{\text{def}}{=} a?.P$, $Q \stackrel{\text{def}}{=} a!.Q$, are non-Zeno processes, but $P \parallel Q$ is not.

The rule for maximal progress is not featured in algebras such as TCCS and ATP. It becomes unnecessary due to the fact that insistent events (recall that the event prefix operator cannot idle) force progress to be made. CSA replaces maximal progress with locally maximal progress appropriate for distributed systems with multiple clocks. As we will see in the next section, ACSR replaces maximal progress with resource-constrained progress.

We continue by reviewing some time-specific constructs introduced in process algebras.

- In addition to the time-prefix operator, TCCS contains an unbounded delay operator $\delta.P$ (note that this is different to the completed process δ) which may delay indefinitely before proceeding with the

computation. This resembles the willingness of patient events to wait for an environment ready to participate. Its behavior is specified by the following two rules:

$$\frac{-}{\delta.P \xrightarrow{\epsilon} \delta.P} \qquad\qquad \frac{P \xrightarrow{e} P'}{\delta.P \xrightarrow{e} P}$$

- In ATP the basic time-passing construct is the binary *unit delay operator*, $\lfloor P \rfloor(Q)$. This process behaves like P if the execution of P starts before the first tick of time takes place. Otherwise, control is passed to process Q. Thus, we may consider the operator as a timeout construct with delay 1. Variants of this operator, including unbounded delay, and bounded but non-unit delay, are encountered in ATP and other algebras.

- Time interrupt constructs, of the form $\lceil P \rceil^d(Q)$, have been defined in ATP and TCSP and describe a process where computation proceeds as for process P for the first d time units, and then Q is started.

- Another useful construct, encountered in TCSP, ATP and ACSR, is the timeout operator. In TCSP and ATP, this is modeled as a binary operator, $P \rhd^d Q$, where d is a positive integer, P is called the body and Q the exception of the timeout. The intuitive meaning of this process is that P is given permission to start execution for d time units, but, if it fails to do so, control is passed to process Q. In ACSR, as we will see below, the timeout operator is more complex and combines the notions of successful termination and interrupts.

Let us now look into the interplay between the above mentioned operators by studying the resulting axioms in various process algebras. First, consider rules (Sum1) and (Par1) of Table 1.1. We observe that, in the presence of axioms (TSUM) and (TPAR), respectively, they do not hold in any of the above-mentioned approaches. However, in the presence of the completed process δ, we have

$$(\mathsf{CSum1}) \quad P + \delta = P \qquad\qquad (\mathsf{CPar1}) \quad P \parallel \delta = P$$

On the other hand, the blocked process NIL satisfies the laws

$$(\mathsf{BSum1}) \quad e.P + \mathrm{NIL} = e.P \qquad\qquad (\mathsf{BPar1}) \quad e.P \parallel \mathrm{NIL} = e.P$$

$$(\mathsf{BSum2}) \quad \epsilon.P + \mathrm{NIL} = \mathrm{NIL} \qquad\qquad (\mathsf{BPar2}) \quad \epsilon.P \parallel \mathrm{NIL} = \mathrm{NIL}$$

When restricting attention to regular processes, that is processes using finitely many guarded recursive equations, an expansion theorem can be deduced for all of the mentioned process algebras.

2.2 Comparative example

In order to tie together the concepts introduced in the previous sections, and to illustrate the effect of design decisions within process-algebraic formalisms on the modeling style they impose, we show two models of the same system in the algebras ATP and TPL. We begin by summarizing the syntax of the two algebras and their associated semantical meaning. To keep presentation simple, we unify the notations of the two formalisms for operators representing the same concept. However, semantic differences between respective constructs may be present and we highlight them by mentioning the semantic rules which apply in each case.

ATP. The syntax of the process algebra ATP is given as follows:

$$P \quad ::= \quad \text{NIL} \;\mid\; C \;\mid\; a.P \;\mid\; P + P \;\mid\; \lfloor P \rfloor(P) \;\mid\; P \parallel P \;\mid\; P \backslash A \;\mid\; \lceil P \rceil^{d}(P)$$

The operators of the language satisfy, among others, the rules (ACT) (and not (WAIT)), (SUM), (TSUM) and (TPAR) (and not (PAR_{MP})). The terminated process δ can be derived in ATP as follows: $\delta \stackrel{\text{def}}{=} \lfloor \text{NIL} \rfloor(\delta)$.

TPL. The syntax of the process algebra TPL is given as follows:

$$P \quad ::= \quad \text{NIL} \;\mid\; C \;\mid\; a.P \;\mid\; \epsilon.P \;\mid\; P + P \;\mid\; \lfloor P \rfloor(P) \;\mid\; P \parallel P \;\mid\; P \backslash A$$

The operators of the language satisfy, among others, the rules (ACT), (WAIT), (SUM), (TSUM) and (PAR_{MP}).

The example. We now proceed to model a simple system in the two algebras. We will show that, despite their syntactic closeness, subtle semantic differences in the definitions of the operators will require us to model the same system differently in order to achieve the same behavior in both languages.

The system under consideration represents two real-time tasks. Task T_1 is a periodic task, which is released every 10 time units and nondeterministically takes either 1 or 2 time units to complete. Once task T_1 completes its execution, it sends a message to release the task T_2, which takes 3 time units to complete. This example has several features that highlight the difference between different process algebras: the treatment of the timeout used in the periodic release of the first task, the choice between time progress and sending an event, necessary to implement the non-deterministic completion time of the first task, and implementation of a lazy event in the second task to receive the release signal from the first task.

We first consider the representation of this system in ATP.

$$\begin{aligned} T_1 &\stackrel{\text{def}}{=} \lceil \lfloor \delta \rfloor (start.\delta + \lfloor \delta \rfloor (start.\delta)) \rceil^{10} (T_1) \\ T_2 &\stackrel{\text{def}}{=} start.\lfloor \delta \rfloor (\lfloor \delta \rfloor (\lfloor \delta \rfloor (T_2))) + \lfloor \delta \rfloor (T_2) \\ System &\stackrel{\text{def}}{=} (T_1 \parallel T_2) \backslash \{start\} \end{aligned}$$

Note that the modeling style of ATP is to describe the events that the process can perform during a given time unit, each within the scope of a unit-delay operator. Consider T_1. In the first time unit, the process is engaged in an internal computation and cannot perform any events. In the second time unit, the process has two choices of behavior. If the internal computation has been completed in the first time unit, the process can perform the event *start* and then idle until the timeout of 10 time units. Otherwise, it idles for another time unit and then has no other choice but to perform the event *start* and wait for the timeout. The process T_2 begins execution when event *start* arrives, with the summand $\lfloor \delta \rfloor (T_2)$ allowing it to idle as needed. Finally, the system is the parallel composition of the two task processes, with the *start* event restricted in order to ensure CSP-style synchronization.

Consider now the same system expressed in TPL. Modeling is distinguished by the ATP description due to ATP's patient events, the explicit time-prefix operator, and the absence of a timeout operator. Furthermore, recall that TPL adopts the CCS style of communication, which results in the use of input and output events for the synchronization of the two tasks.

$$\begin{aligned} T_1 &\stackrel{\text{def}}{=} \epsilon.(start!.\epsilon^9.T_1 + \tau.\epsilon.start!.\epsilon^8.T_1) \\ T_2 &\stackrel{\text{def}}{=} start?.\epsilon.\epsilon.\epsilon.T_2 \\ System &\stackrel{\text{def}}{=} (T_1 \parallel T_2) \backslash \{start\} \end{aligned}$$

Here, for simplicity, we employ the notation $e^n.P$ for the process that makes n consecutive executions of event e before proceeding to process P. Thus, task T_1 initially performs some internal computation during the first time unit, and then it evolves into a choice process. This presents the two possibilities for the duration of the execution of the task. The first summand corresponds to the case the task is completed after a single time unit, in which case a *start* message is emitted. If not, an internal action is performed by the task and, after a second time unit elapses, task T_2 is notified and the process idles for a further 8 time

units. Note that, unlike the ATP model, here it is not necessary to model explicitly the waiting of task T_2: the process waits indefinitely until the action becomes enabled. However, due to maximal progress, once the communication on the channel *start* becomes enabled, it cannot be further delayed by time passage. Consequently, the τ action included in the description of T_1 is crucial for the correct description of the model. Had it been absent, maximal progress would enforce the communication on *start* to take place immediately in a process such as $(start!.P + \epsilon.Q \parallel start?.R)$.

3 Modeling of Resource-Sensitive Systems

The notion of maximal progress employed by many real-time process algebras is a useful modeling concept. Maximal progress makes the model engage in useful computation whenever it is able to do so. However, there are many cases when maximal progress does not fit well with the application domain. Many systems that operate under real-time constraints are also constrained by available computational resources. This is especially true of embedded systems that have to operate on processors with limited computing power and small memory. Clearly, satisfaction of timing constraints by such systems depends on timely availability of resources needed by each process. Therefore, modeling of resource-constrained systems replaces the notion of maximal progress with the notion of resource-constrained progress.

Process algebra ACSR [11] implements this notion of resource-constrained progress by associating resource consumption with time passage. Specifically, it replaces the single timed event ϵ, discussed in the previous section, by a structured set of labels: whenever a process wants to perform a step that advances time, it declares the set of resources that are needed for this step, along with the priority of access for each resource. This resource declaration is called an *action*. The semantic rules of the parallel composition operator ensure that only actions using disjoint resources are allowed to proceed simultaneously. Thus, while progress in the system is enforced wherever possible, in order for two processes to consume the same resource two distinct phases (time units) must elapse. We will build upon the discussion in Section 2 to highlight similarities and differences between ACSR and other real-time process algebras.

3.1 Syntax and semantics

Execution steps. ACSR employs a different execution model compared to other real-time process algebras. We have previously seen that processes in most process algebras execute events that are considered instantaneous and, in between events, time can advance while processes are "doing nothing." In ACSR, processes can also engage in instantaneous events. The CCS-style of communications is adopted for such events which can be observable, such as sending or receiving a message, or the internal τ action. On the other hand, it assumes that processes can engage in internal computation that takes non-trivial amount of time relative to the size of the clock tick. These timed actions involve the consumption of resources, or explicit idling. Consequently, ACSR distinguishes between events and actions. Just like the events that we have considered above, ACSR events are logically instantaneous and do not require computational resources to execute. An action, by contrast, takes one unit of time and requires a set of resources to complete. We consider the details below.

Timed Actions We consider a system to be composed of a finite set of serially-reusable resources denoted by $\mathcal{R}$. An action that consumes one "tick" of time is a set of pairs of the form (r, p), where r is a resource and p, a natural number, is the priority of the resource use, with the restriction that each resource be represented at most once. As an example, the singleton action, $\{(cpu, 2)\}$, denotes the use of some resource $cpu \in \mathcal{R}$ running at priority level 2. The action $\emptyset$ represents idling for one time unit, since no resource is consumed.

We use $\mathcal{D}_R$ to denote the domain of timed actions, and we let A, B range over $\mathcal{D}_R$. We define $\rho(A)$ to be the set of resources used in the action A; e.g., $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$.

Instantaneous Events Instantaneous events provide the basic synchronization mechanism in ACSR. We assume a set of channels L. An event is denoted by a pair (a, p), where a is the *label* of the event, and p is its *priority*. Labels are drawn from the set $\text{Ł} \cup \bar{\text{Ł}} \cup \{\tau\}$, where for all $a \in L$, $a? \in \text{Ł}$ and $a! \in \bar{\text{Ł}}$. We say that $a?$ and $a!$ are *inverse* labels. As in CCS, the special identity label τ arises when two events with inverse labels are executed in parallel.

We use $\mathcal{D}_E$ to denote the domain of events, and let e, f range over $\mathcal{D}_E$. We use $l(e)$ to represent the label of event e. The entire domain of actions is $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$, and we let α and β range over $\mathcal{D}$.

ACSR operators. The following grammar describes the syntax of PACSR processes:

$$P \quad ::= \quad \text{NIL} \mid A : P \mid e.P \mid P + P \mid P \parallel P \mid$$
$$P \triangle_t^a (P, P, P) \mid P \backslash F \mid [P]_I \mid P \backslash\backslash I \mid C$$

Semantics for ACSR is set up in two steps. First, the *unprioritized* semantics define steps of a process by means of structural rules, as we have seen with the other process algebras above. Second, the *prioritized* semantics interpret priorities within events and actions by applying a *preemption relation*, disabling low-priority events and actions in the presence of higher-priority ones. The rules of the unprioritized semantics are given in Table 1.2.

Process NIL is a deadlocked process that blocks the progress of time. Event prefix $e.P$ is similar to the event prefix of ATP and cannot pass time. Action prefix $A : P$ is similar to the time prefix of TPL, except that a distinct action A is executed instead of ϵ. Process $P + Q$ offers the nondeterministic choice between P and Q. This choice is resolved with the first action taken by one of the two processes.

The process $P \parallel Q$ describes the concurrent composition of P and Q: the component processes may proceed independently or interact with one another while executing events, and they synchronize on timed actions. Specifically, rules (Par1)-(Par3) implement CCS-style communication in the model, with the first two rules describing the asynchronous execution of events, and the latter describing synchronization on matching events. Of particular interest to the algebra is rule (Par4). We first observe that time progresses synchronously in a complex system, that is, for time to pass all of the concurrent components must be willing to perform a timed action. However, note that such progress is resource-constrained: as expressed in the side condition of the rule, at most one process may use a given resource in any time unit. A consequence of this side condition is that whenever two (or more) concurrent processes are competing for the use of the same resource and neither is willing to engage in alternative behariour, then the system is deadlocked. This fact plays a significant role in the algebra and it is exploited for performing schedulability analysis. For example, process $\{(cpu, 1), (mem, 2)\} : P \| \{(cpu, 2)\} : Q$ has no outgoing transitions since $\{(cpu, 1), (mem, 2)\} \cap \{(cpu, 2)\} \neq \emptyset$. On the other hand:

$$\{(cpu_1, 1), (mem, 2)\} : P \parallel (\{(cpu_1, 2)\} : Q_1 + \{cpu_2, 1\} : Q_2) \xrightarrow{\{(cpu_1,1),(mem,2),(cpu_2,1)\}} P \parallel Q_2$$

The scope construct, $P \triangle_t^a (Q, R, S)$, binds the process P by a temporal scope and incorporates the

Table 1.2: **The non-prioritized relation**

(Act1)	$e.P \xrightarrow{e} P$	(Act2)	$A : P \xrightarrow{A} P$
(Sum1)	$\dfrac{P_1 \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P}$	(Sum2)	$\dfrac{P_2 \xrightarrow{\alpha} P}{P_1 + P_2 \xrightarrow{\alpha} P}$
(Par1)	$\dfrac{P_1 \xrightarrow{e} P_1'}{P_1 \| P_2 \xrightarrow{e} P_1' \| P_2}$	(Par2)	$\dfrac{P_2 \xrightarrow{e} P_2'}{P_1 \| P_2 \xrightarrow{e} P_1 \| P_2'}$
(Par3)	$\dfrac{P_1 \xrightarrow{(a?,n)} P_1',\ \ P_2 \xrightarrow{(a!,n)} P_2'}{P_1 \| P_2 \xrightarrow{(\tau,n+m)} P_1' \| P_2'}$		
(Par4)	$\dfrac{P_1 \xrightarrow{A_1} P_1',\ P_2 \xrightarrow{A_2} P_2'}{P_1 \| P_2 \xrightarrow{A_1 \cup A_2} P_1' \| P_2'}, \quad \rho(A_1) \cap \rho(A_2) = \emptyset$		
(Res1)	$\dfrac{P \xrightarrow{e} P',\ \ l(e) \notin F}{P \backslash F \xrightarrow{e} P' \backslash F}$	(Res2)	$\dfrac{P \xrightarrow{A} P'}{P \backslash F \xrightarrow{A} P' \backslash F}$
(Cl1)	$\dfrac{P \xrightarrow{A_1} P', A_2 = \{(r,0) \mid r \in I - \rho(A_1)\}}{[P]_I \xrightarrow{A_1 \cup A_2} [P']_I}$	(Cl2)	$\dfrac{P \xrightarrow{e} P'}{[P]_I \xrightarrow{e} [P']_I}$
(Hide1)	$\dfrac{P \xrightarrow{A} P',\ \ A' = \{(r,n) \in A \mid r \notin I\}}{P \backslash\backslash I \xrightarrow{A'} P' \backslash\backslash I}$	(Hide2)	$\dfrac{P \xrightarrow{e} P'}{P \backslash\backslash I \xrightarrow{e} P' \backslash\backslash I}$
(Sc1)	$\dfrac{P \xrightarrow{e} P',\ \ l(e) \neq b!,\ \ t > 0}{P \triangle_t^b (Q,R,S) \xrightarrow{e} P' \triangle_t^b (Q,R,S)}$	(Sc2)	$\dfrac{P \xrightarrow{(b!,n)} P',\ \ t > 0}{P \triangle_t^b (Q,R,S) \xrightarrow{(\tau,n)} Q}$
(Sc3)	$\dfrac{P \xrightarrow{A} P',\ \ t > 0}{P \triangle_t^b (Q,R,S) \xrightarrow{A} P' \triangle_{t-1}^b (Q,R,S)}$	(Sc4)	$\dfrac{R \xrightarrow{\alpha} R',\ \ t = 0}{P \triangle_t^b (Q,R,S) \xrightarrow{\alpha} R'}$
(Sc5)	$\dfrac{S \xrightarrow{\alpha} S',\ \ t > 0}{P \triangle_t^b (Q,R,S) \xrightarrow{\alpha} S'}$	(Rec)	$\dfrac{P \xrightarrow{\alpha} P', C \stackrel{\text{def}}{=} P}{C \xrightarrow{\alpha} P'}$

notions of timeout and interrupts. We call t the *time bound*, where $t \in \mathbb{N} \cup \{\infty\}$ and require that P may execute for a maximum of t time units. The scope may be exited in one of three ways: first, if P terminates successfully within t time-units by executing an event labeled $a!$, where $a \in L$, then control is delegated to Q, the success-handler. Else, if P fails to terminate within time t, then control proceeds to R. Finally, throughout the execution of this process, P may be interrupted by process S.

As an example consider the task specification $T \stackrel{\text{def}}{=} R \triangle^{a}_{10} (SH, EH, IN)$ where

$$\begin{aligned} R &\stackrel{\text{def}}{=} (in?, 1).(a!, 2).\text{NIL} + \emptyset : R \\ SH &\stackrel{\text{def}}{=} (ack!, 1).T \\ EH &\stackrel{\text{def}}{=} (nack!, 1).T \\ IN &\stackrel{\text{def}}{=} (kill?, 3).\text{NIL} \end{aligned}$$

This task awaits for an input request to arrive for a 10 time-unit period. If such an event takes place it signals on channel a the arrival and the success handler process, SH, acknowledges the event. If the deadline elapses without the appearance of the event, the task signals the lack of input on channel *nack*. Finally, at any point during its computation, the task may receive a signal on channel *kill* and halt its computation. According to the rules for scope, process $R \triangle^{a}_{10} (SH, EH, IN)$ may engage in the following actions:

$$\begin{aligned} R \triangle^{a}_{10} (SH, EH, IN) &\xrightarrow{\emptyset} R \triangle^{a}_{9} (SH, EH, IN) \\ R \triangle^{a}_{10} (SH, EH, IN) &\xrightarrow{(in?,1)} ((a!, 2).\text{NIL}) \triangle^{a}_{10} (SH, EH, IN) \\ R \triangle^{a}_{10} (SH, EH, IN) &\xrightarrow{(kill?,3)} \text{NIL} \end{aligned}$$

Furthermore, note that:

$$\begin{aligned} R \triangle^{a}_{0} (SH, EH, IN) &\xrightarrow{(nack!,1)} T \\ ((a!, 2).\text{NIL}) \triangle^{a}_{10} (SH, EH, IN) &\xrightarrow{(\tau,2)} SH \end{aligned}$$

The Close operator, $[P]_I$, produces a process P that monopolizes the resources in $I \subseteq \mathcal{R}$. Rules (Cl1) and (Cl2) describe the behavior of the close operator. When a process P is embedded in a closed context such as $[P]_I$, we ensure that there is no further sharing of the resources in I. Assume that P executes a time-consuming action A. If A utilizes less than the full resource set I, the action is augmented with $(r, 0)$ pairs for each unused resource $r \in I - \rho(A)$. The way to interpret Close is as follows. A process may idle in two ways: it may either release its resources during the idle time (represented by $\emptyset$), or it may hold them. Close ensures that the resources are held. Instantaneous events are not affected. Thus:

$$[\emptyset : P_1 + \{(cpu, 1)\} : P_2]_{cpu} \xrightarrow{\{(cpu,0)\}} P_1$$

$$[\emptyset : P_1 + \{(cpu, 1)\} : P_2]_{cpu} \quad \overset{\{(cpu,1)\}}{\longrightarrow} \quad P_2$$

The resource hiding operator, $P\backslash\backslash I$ internalizes the set of resources I within the process P. As shown in (Hide1) and (Hide2) information regarding usage of resources in I is removed from timed actions of P while instantaneous events remain unaffected. Finally, the restriction operator $P\backslash F$ and process constants C are defined in the usual way.

In subsequent sections, we will assume the presence of a special constant $IDLE \in \mathcal{C}$, representing the process that idles forever, defined by $IDLE \overset{\text{def}}{=} \emptyset : IDLE$. Further, we will be using the shorthand notation $P \triangle_t (Q)$ for $P \overset{a}{\triangle_t} (\text{NIL}, Q, \text{NIL})$ to denote the process that executes as P for the first t time units and then proceeds as Q. Finally, we write $\Pi_{i \in I} P_i$ for the parallel composition of processes P_i, $i \in I$.

An important observation to make about the semantic rules is that the only "source" of time progress is the action-prefix operator. That is, time progress has to be explicitly encoded into the model by the designer. This feature forces the designer to think carefully about time progress in the model and results in fewer unexpected behaviors in a complex model.

Another important observation is that time determinism, which is one of the main underlying principles for most real-time process algebras, loses its importance in ACSR. Specifically, time determinism requires that if $P \xrightarrow{\epsilon} P_1$ and $P \xrightarrow{\epsilon} P_2$, then $P_1 = P_2$. That is, because a process is, effectively, idle while time advances, one tick should be indistinguishable from another. On the other hand, in ACSR, a process may be able to perform different timed actions and evolve to different states: $P \xrightarrow{A_1} P_1$ and $P \xrightarrow{A_2} P_2$, with $P_1 \neq P_2$. Since actions may be blocked by other processes with higher resource-access priorities, it is important to develop models in such a way that a process offers alternative behaviors in case the main one is blocked. For example, a process P that wants to execute an action A, but is willing to be preempted, can be expressed as $P \overset{\text{def}}{=} A : P' + \emptyset : P$. The action $\emptyset$, which does not require any resources, allows the process P to idle for one time unit and then retry to execute A.

Preemption and Prioritized Transitions. The prioritized transition system is based on *preemption*, which incorporates the ACSR treatment of priority. This is based on a transitive, irreflexive, binary relation on actions, $\prec$, called the *preemption relation*. If $\alpha \prec \beta$, for two actions α and β, we say that α *is preempted*

by β. Then, in any process, if there is a choice between executing either α or β, β will always be executed. We refer to [11] for the precise definition of $\prec$. Here, we briefly describe the three cases for which $\alpha \prec \beta$ is deemed to be true by the definition.

- The first case is for two timed actions α and β which compete for common resources. Here, it must be that the preempting action β employs all its resources at priority level at least the same as α. Also, β must use at least one resource at a higher level. It is still permitted for α to contain resources not in β but all such resources must be employed at priority level 0. Otherwise, the two timed actions are incomparable. Note that β cannot preempt an action α consuming a *strict subset* of its resources at the same or lower level. This is necessary for preserving the compositionality of the parallel operator. For instance, $\{(r_1, 2), (r_2, 0)\} \prec \{(r_1, 7)\}$ but $\{(r_1, 2), (r_2, 1)\} \not\prec \{(r_1, 7)\}$.

- The second case is for two events with the same label. Here, an event may be preempted by another event with the same label but a higher priority. For example, $(\tau, 1) \prec (\tau, 2)$, $(a, 2) \prec (a, 5)$, and $(a, 1) \not\prec (b, 2)$ if $a \neq b$.

- The third case is when an event and a timed action are comparable under "$\prec$." Here, if $n > 0$ in an event (τ, n), we let the event preempt any timed action. For instance, $\{(r_1, 2), (r_2, 5)\} \prec (\tau, 2)$, but $\{(r_1, 2), (r_2, 5)\} \not\prec (\tau, 0)$.

We define the prioritized transition system "$\rightarrow_\pi$," which simply refines "$\rightarrow$" to account for preemption as follows: $P \xrightarrow{\alpha}_\pi P'$ if and only if (1) $P \xrightarrow{\alpha} P'$ is an unprioritized transition, and (2) there is no unprioritized transition $P \xrightarrow{\beta} P''$ such that $\alpha \prec \beta$.

We note that the close operator has a special significance in relation to the preemption relation and the prioritized transition system. In particular, we may see that, by closing a system by the set of its resources, we enforce progress to be made: for example, consider the case when an idling action and a resource-consuming action are simultaneously enabled from the initial state of a system. By applying the close operator, the idling action is transformed into an action that consumes all resources of the system at priority level 0. Then, in the prioritized transition relation this action will become preempted by the non-idling action, and the system will be forced to progress.

3.2 Schedulability Analysis with ACSR

In this section, we discuss how realistic task sets can be modeled in ACSR and how to perform schedulability analysis of such task sets. A *real-time task* is a building block for a real-time system. A task is invoked in response to an event in the system and needs to perform its computation by a certain deadline. A task invocation, then, is characterized by two parameters that are constant across all invocations of the same task: a relative deadline d and the range of execution times it takes to complete, $[c_{min}, c_{max}]$. The task itself can be considered as a sequence of invocations and is characterized by a *dispatch policy*, which can be periodic, aperiodic, or sporadic. A periodic task with a period p is dispatched by a timer event every p time units. Aperiodic and sporadic tasks are dispatched by events that can be raised by tasks within the system or originate from the system's environment. Sporadic tasks are characterized by the minimum separation between task invocations. For more information the reader is referred to [12]. A real-time system consists of a fixed set of tasks and a set of resources that tasks share. The most important kind of resources are processors that run tasks. Although ACSR can model migration of tasks between processors, in this paper we assume that tasks are statically allocated to processors. Other resource types include buses, shared memory blocks, etc.

In [5], modeling and analysis of several scheduling problems was introduced. Here, we apply a similar approach to a somewhat different scheduling problem. We consider real-time systems that contain periodic tasks with precedence constraints, executing on a set of processor resources without task migration. These tasks are competing between them for the use of the system's resources, access to which is resolved by a scheduler according to the task priorities. These priorities can be static or dynamic. We assume that the systems we consider are *closed*, that is, there is no competition for system resources from outside sources. Similar task models have been considered in [18, 20]. We use the following notation: For a task T_i, let the integer constants d_i, $[c_{min,i}, c_{max,i}]$, and p_i, denote the deadline, execution time range, and period of the task, respectively. Resource cpu_i denotes the processor resource to which T_i is allocated. Note that, if more than one tasks are allocated to a processor, then $cpu_i = cpu_j$ for some i and j.

We assume a set of precedence constraints between the system tasks representing data or control depen-

Figure 1.1: Task set with precedence constraints

dencies that exist between them. These constraints are of the form

$$w_{ij} : T_i \overset{f_{ij}}{\twoheadrightarrow} T_j,$$

where f_{ij} is an integer constant representing a propagation delay. We require that the precedence graph be acyclic in order to avoid deadlocks during execution. We also use the simplifying assumption that for every constraint w_{ij} tasks T_i and T_j have harmonic periods; that is, either p_i is a multiple of p_j or vice versa. Let $t_{j,n}$ be the n^{th} invocation of task T_j. Then, a constraint $w_{ij} : T_i \overset{f_{ij}}{\twoheadrightarrow} T_j$ with $p_j = k \cdot p_i$ (that is, T_i executes more often than T_j), is interpreted as follows: $t_{j,n}$ cannot begin execution until f_{ij} time units later than $t_{i,(n-1)\cdot k+1}$ completes its execution. On the other hand, if $w_{ij} : T_i \overset{f_{ij}}{\twoheadrightarrow} T_j$ with $p_i = k \cdot p_j$, then $t_{j,n}$ cannot begin execution until f_{ij} time units later than $t_{j,n \bmod k}$ completes its execution.

As we present the modeling approach, it will become clear that this task model can be easily generalized in several ways, at the cost of a slightly more complex ACSR encoding. More general constraints can be used, the requirement for harmonic periods for interacting tasks can be eliminated, and other dispatch policies can be represented. We also note that the encoding of tasks and constraints can be automatically generated from higher-level architectural representations such as AADL, as reported in [23].

Example. To illustrate the scheduling problem described above, consider the example shown in Figure 1.1(a). The system consists of four tasks with constant execution times ($c = c_{min} = c_{max}$) and deadlines equal to task periods. Constraints are represented as a graph, with tasks as graph nodes and constraints as directed edges. Figure 1.1(b) presents the schedule of a system execution with the rate-monotonic priority assignment. (Tasks are assigned static priorities according to their periods, the shorter the period the higher the priority, and scheduling is preemptive.) Note that even though task T_4 has the highest priority, it barely meets its deadline due to a precedence dependency on a lower-priority T_2, which is preempted by T_1. Note also that only the first invocation of T_4 is affected by T_2, and only the first invocation of T_1 affects T_3.

We now proceed with the model of the scheduling problem. We do this in a bottom-up approach, beginning with the modeling of tasks and building up to the modeling of an entire system.

Encoding of task execution. The execution of a task T_i goes through several states. Initially, T_i is sleeping, represented by the ACSR process $Sleep_i$. This process can idle indefinitely, waiting for the environment to offer the $dispatch_i$ event. When this event occurs, the process proceeds to the next state $Resolve_i$, where it waits until all precedence constraints are met. When that happens, the environment offers the $csat_i$ event, which will allow the task to begin its execution. Execution is modeled by a parameterized collection of ACSR processes $Exec_{i,c}$. Intuitively, process $Exec_{i,c}$ represents the execution when the task invocation has accumulated c time units of computation. While $c < c_{i,max}$, $Exec_{i,c}$ can gain access to the resource cpu_i, and by doing this it advances to $Exec_{i,c+1}$. Alternatively, the task may be preempted, in which case $Exec_{i,c}$ follows the self-loop labeled with the idling action. In addition, if $c >= c_{i,min}$, $Exec_{i,c}$ can send event $done_i$ and return to state $Sleep_i$. Once $c = c_{i,max}$, the process may only offer the event $done_i$ until it becomes accepted. The ACSR processes for the task execution are shown below:

$$
\begin{aligned}
Sleep_i &\stackrel{\text{def}}{=} \emptyset : Sleep_i + (dispatch_i?, 0).Resolve_i \\
Resolve_i &\stackrel{\text{def}}{=} \emptyset : Resolve_i + (csat_i?, 0).Exec_{i,0} \\
Exec_{i,c} &\stackrel{\text{def}}{=} \emptyset : Exec_{i,c} + \{(cpu_i, pr_i)\} : Exec_{i,c+1} && c < c_{i,min} \\
Exec_{i,c} &\stackrel{\text{def}}{=} \emptyset : Exec_{i,c} + \{(cpu_i, pr_i)\} : Exec_{i,c+1} + (done_i!, pr_i).Sleep_i && c_{i,min} \leq c < c_{i,max} \\
Exec_{i,c} &\stackrel{\text{def}}{=} \emptyset : Exec_{i,c} + (done_i!, pr_i).Sleep_i && c = c_{i,max}
\end{aligned}
$$

Task activators. For each task T_i, the event $dispatch_i$ is supplied by a process acting as a *task activator*. The activator process serves two purposes: it handles thread dispatch and keeps track of the deadline of the current thread invocation. Here we show a simple activator process for a periodic thread T_i whose deadline is equal to its period ($d_i = p_i$).

$$
\begin{aligned}
Activator_i &\stackrel{\text{def}}{=} (dispatch_i!, pr_i).Wait \triangle_{p_i} (Activator_i) \\
Wait &\stackrel{\text{def}}{=} (done?, pr_i).Signal_i + \emptyset : Wait
\end{aligned}
$$

The process sends the dispatch event to the task process, initiating execution of the task invocation. Then, the activator awaits for p_i time units for the task to signal its completion on channel *done* in which case it continues as process $Signal_i$, explained in the next paragraph. Note that, as soon as the task is ready to emit an event labeled with *done* this will be immediately received by *Wait*. This is ensured by the

preemption relation that enforces maximal progress in this sense by preempting timed actions over internal communications. Once the p_i time units elapse, the $Activator_i$ resends the dispatch signal. By that time, the thread should have completed its activation and be able to accept the dispatch signal. Otherwise, a deadline violation is detected and the activator process is blocked since, in its initial state, it cannot pass time. The resulting deadlocked state in the state space of the model allows us to detect the missed deadline.

Encoding of constraints. The set of precedence constraints of the system is handled by associating two more processes with each task in the model. The first is responsible for *releasing* all tasks which are waiting for the task to complete execution, and the latter is for detecting the release of all constraints on which its own invocation depends. So, suppose that task T_i is involved in a set of constraints $W_1 = \{w_{i,j} \mid j \in J\}$, and a set of constraints $W_2 = \{w_{m,i} \mid m \in M\}$. Let us begin by considering constraints W_1. For each such constraint one should detect the completion of task T_i and send appropriate messages to release the relevant task. The detection of the completion is implemented by process $Signal_i$, the component of $Activator_i$ mentioned above, as follows:

$$Signal_i \stackrel{\text{def}}{=} (\mathit{fin}_i!, pr_i).Signal_i + \emptyset : Signal_i$$

This process is enabled until the next invocation of $Task_i$ and emits the event fin_i signaling the completion of an invocation. Such a completion may have different importance for difference constraints. To capture this, for each constraint $w_{i,j} \in W_1$, we define a process $Release_{i,j}$ that determines whether the constraint should be released and emits a message to task j. Representation of $Release_{i,j}$ depends on the relationship between the periods of T_i and T_j. In the case that $p_j = k \cdot p_i$, $Release_{i,j}$ awaits the completion of task T_i and emits a message to task T_j by offering $done_{i,j}$. It repeats this behavior every p_j time units and ignores intermediate completions of invocations:

$$\begin{aligned} Release_{i,j} &\stackrel{\text{def}}{=} Rel_{i,j} \triangle_{p_j} (Release_{i,j}) \\ Rel_{i,j} &\stackrel{\text{def}}{=} (\mathit{fin}_i?, pr_i).Send_{i,j} + \emptyset : Rel_{i,j} \\ Send_{i,j} &\stackrel{\text{def}}{=} (done_{i,j}!, pr_i).IDLE + \emptyset : Send_{i,j} \end{aligned}$$

The case of $p_i = k \cdot p_j$ is very similar with the exception that $Release_{i,j}$ repeats this behavior every p_i time units.

Let us now consider constraints W_2 for a task T_i. These are handled by an ACSR process $Constr_i$. The process itself consists of a number of concurrent processes: a set of processes $Prop_{j,i}$, for each $w_{j,i} \in W_2$, which detect the release of the constraint $w_{j,i}$, and a process $Poll_i$ which queries all the $Prop_{j,i}$ processes and sends the $csat_i$ event if all constraints are released. Communication between these concurrent processes is internal to $Constr_i$

$$Constr_i \stackrel{\text{def}}{=} (Poll_i \ \| \ \Pi_j Prop_{i,j}) \backslash \{ready_i\}$$

The process $Poll_i$ runs every time unit and sequentially tries to synchronize with every process $Prop_{j,i}$ on event $ready_i$. If any of the processes cannot offer $ready_i$, this indicates that the constraint $w_{j,i}$ is not satisfied. Let us assume that $|W_2| = m$. We have:

$$\begin{aligned}
Poll_i &\stackrel{\text{def}}{=} Poll_{i,0} \\
Poll_{i,j} &\stackrel{\text{def}}{=} (ready_i?, 0).Poll_{i,j+1} + \emptyset : Poll_i \quad j < m \\
Poll_{i,m} &\stackrel{\text{def}}{=} (csat_i!, pr_i).\emptyset : Poll_i + \emptyset : Poll_i.
\end{aligned}$$

Representation of $Prop_{i,j}$ depends on the relationship between the periods of T_i and T_j. Let us begin with the case that $p_i = k \cdot p_j$. In this case, $Prop_{i,j}$ awaits a signal from process $Release_{j,i}$ and, after $f_{j,i}$ time units, it declares the release of the task to the polling process by sending the event $ready_i$. It repeats this behavior every p_i time units:

$$\begin{aligned}
Prop_{i,j} &\stackrel{\text{def}}{=} P_{i,j} \triangle_{p_i} (Prop_{i,j}) \\
P_{i,j} &\stackrel{\text{def}}{=} (done_{j,i}?, pr_i).IDLE \triangle_{f_{j,i}} (Ready_{i,j}) + \emptyset : P_{i,j} \\
Ready_{i,j} &\stackrel{\text{def}}{=} \emptyset : Ready_{i,j} + (ready_j!, pr_{i,l}).Ready_{i,j}
\end{aligned}$$

The case when $p_j = k \cdot p_i$ is handled in a similar way, only the behavior of $Prop_{i,j}$ is repeated every p_j time units.

Priority assignments and final composition. We are now ready to put together the processes for tasks and constraints into a coherent model. We first form the "task cluster" that brings together the processes that relate to the single task, and make communication between them internal to the cluster. The four processes involved in a cluster for the task T_i are $Sleep_i$, $Activator_i$, $Release_i$, and $Constr_i$. They communicate by

events $dispatch_i$, $done_i$, fin_i, and $csat_i$. We thus get the following expression for the task cluster process

$$Task_i \stackrel{\text{def}}{=} (Sleep_i \| Activator_i \| Constr_i \| Release_i) \backslash \{dispatch_i, done_i, csat_i, fin_i\}.$$

It remains to compose task clusters together, and make all communication on events $done_{i,j}$ internal. In addition, we need to close all processor resources so that no other tasks can use them and to enforce progress in the system.

$$RTSystem \stackrel{\text{def}}{=} [(\Pi_{i=1\ldots n} Task_i) \backslash \{done_{i,j} | i,j = 1 \ldots n\}]_{\{cpu_i | i=1\ldots n\}}.$$

In order for this model to execute correctly, we need to carefully assign priorities in actions and events used in the model. Priorities of resource access represent the scheduling policy used to schedule processors in the system. In the case of static priority scheduling, all actions in all processes $Exec_{i,c}$ have the same priority, which is the priority statically assigned to thread T_i. Dynamic-priority scheduling policies, such as EDF (Earliest Deadline First) policy, makes the resource priority in $Exec_{i,c}$ a function of d_i and c.

On the other hand, event priorities do not play an important role in the specific model. We may observe that any process may engage in at most one event at any given point in time. Thus, although the execution of more than one internal event may be possible in any time-phase (between two timed actions) in the global system, if a number of internal events are simultaneously enabled in some state of the system, execution of one event does not affect the execution of another, since, by construction, this must be taking place in a separate component of the system. In addition, note that all such events will take place confluently *before* the next timed action due to the preemption relation preempting timed events by internal actions.

Schedulability Analysis Within the ACSR formalism we can conduct two types of analysis for real-time scheduling: validation and schedulability analysis. Validation shows that a given specification correctly models the required real-time scheduling discipline, such as Rate Monotonic and Earliest-Deadline-First. Schedulability analysis determines whether or not a real-time system with a particular scheduling discipline misses any of its deadlines. In the ACSR modeling approach, a missed deadline induces a deadlock in the system. The correctness criterion, then, can be stated as the absence of deadlocks in the model of the system. This can be checked by using state-space exploration techniques. Specifically, we can show that an instance of the problem is schedulable if and only if its associated *RTSystem* process contains no deadlocks. The

following theorem pinpoints the source of deadlocks in a certain class of systems, including process *RTSystem* above, and enables ut to arrive at the desired conclusion.

First, let us introduce some useful notation. In the sequel, we will write $P \xrightarrow{\alpha}_\pi$ if there is some process P' such that $P \xrightarrow{\alpha}_\pi P'$ and $P \not\xrightarrow{\alpha}_\pi$ if there is no process P' such that $P \xrightarrow{\alpha}_\pi P'$. Further, we will write $P \Longrightarrow_\pi P'$ if there is a sequence of transitions of the form $P \xrightarrow{\alpha_1}_\pi P_1 \xrightarrow{\alpha_2}_\pi \ldots \xrightarrow{\alpha_n}_\pi P_n = P'$, in which case we say that P' is a *derivative* of P and that the *length* of the transition $P \Longrightarrow_\pi P'$ is n.

Theorem 1.1 *Consider a process P such that*

$$P \stackrel{\text{def}}{=} ([(\Pi_{i\in I} P_i)\backslash F]_U)\backslash\backslash V$$

where $F \subseteq L$, $U, V \subseteq \mathcal{R}$, for all i, P_i contains no parallel composition operator and $I = I_1 \cup I_2$ where,

- *for all $i \in I_1$ and for all derivatives Q_i of P_i, $Q_i \xrightarrow{\emptyset}_\pi$, and*
- *for all $i \in I_2$ and for all derivatives Q_i of P_i, either (1) $Q_i \xrightarrow{\emptyset}_\pi$ or (2) $Q_i \xrightarrow{\alpha_i}_\pi$, $\ell(\alpha_i) \in F$, and $Q_i \not\xrightarrow{\beta}_\pi$ for all $\beta \in \mathcal{D} - \{\alpha_i\}$.*

Then, for all derivatives Q of P, if $Q \not\longrightarrow_\pi$ then $Q = ([(\Pi_{i\in I} Q_i)\backslash F]_U)\backslash\backslash V$ and $Q_i \xrightarrow{\alpha_i}_\pi$ for some $i \in I_2$.

PROOF: Consider a process $P \stackrel{\text{def}}{=} ([(\Pi_{i\in I} P_i)\backslash F]_U)\backslash\backslash V$ satisfying the conditions of the theorem. We will prove that any derivative Q of P is such that $Q = [(\Pi_{i\in I} Q_i)\backslash F]_U)\backslash\backslash V$, where no Q_i contains a parallel composition operator, and I can be partitioned into sets I_1 and I_2 satisfying the conditions of the theorem. The proof will be carried out by induction on the length, n, of the transition $P \Longrightarrow_\pi Q$.

Clearly, the claim holds for $n = 0$. Suppose that it holds for $n = k - 1$ and that $P \Longrightarrow_\pi Q' \xrightarrow{\alpha}_\pi Q$ is a transition of size n. By the induction hypothesis, $Q' = [(\Pi_{i\in I} Q'_i)\backslash F]_U)\backslash\backslash V$ satisfies the conditions of the theorem. Consider the transition $Q' \xrightarrow{\alpha}_\pi Q$. Three cases exist:

- $\alpha \in \mathcal{D}_R$. This implies that for all $i \in I$, $Q'_i \xrightarrow{A_i}_\pi Q_i$, for some Q_i, $Q = [(\Pi_{i\in I} Q_i)\backslash F]_U)\backslash\backslash V$ and $\alpha = \bigcup_{i\in I} A_i$. It is straightforward to see that no Q_i contains a parallel composition operator and that, since each Q_i is a derivative of Q'_i, the conditions of the theorem are satisfied.
- $\alpha = \tau$. This implies that there exist $j, k \in I$, such that $Q'_j \xrightarrow{\alpha_j}_\pi Q_j$ and $Q'_k \xrightarrow{\alpha_k}_\pi Q_k$, where $\ell(\alpha_j)$ and $\ell(\alpha_k)$ are inverse labels, and $Q = [(\Pi_{i\in I-\{j,k\}} Q'_i \parallel Q_j \parallel Q_k)\backslash F]_U)\backslash\backslash V$. It is straightforward to see

that no Q'_i, Q_i, contains a parallel composition operator and check that the conditions of the theorem are satisfied.

- $\alpha \in \mathcal{D}_E$. This implies that there exists $j \in I$, such that $Q'_j \xrightarrow{\alpha_j}_\pi Q_j$ $Q = [(\Pi_{i \in I - \{j\}} Q'_i \parallel Q_j) \backslash F]_U) \backslash\backslash V$ and the proof follows easily.

So consider an arbitrary derivative Q of P and suppose that $Q \not\longrightarrow_\pi$. Since $Q = [(\Pi_{i \in I} Q_i) \backslash F]_U) \backslash\backslash V$ satisfies the conditions of the theorem, and further $Q \not\xrightarrow{\emptyset}_\pi$, it must be that some $Q_i \not\xrightarrow{\emptyset}_\pi$, $i \in I_2$. This implies that $Q_i \xrightarrow{\alpha_i}_\pi$ and the result follows. □

As a corollary we obtain the desired result.

Proposition 1.1 *RTSystem is schedulable if and only if it contains no deadlocks.*

PROOF: First, we observe that if *RTSystem* contains no deadlocks then the associated real-time system is schedulable: task activations take place as planned and no deadlines are missed.

To prove the opposite direction, we show that if the system contains a deadlock then *RTSystem* is not schedulable. Consider system *RTSystem*. By using the ACSR axiom system, we may easily rewrite this process to an equivalent process which has the form of P in Theorem 1.1, that is,

$$RTSystem = [\ (\Pi_{i=1\ldots n}\ \ (Sleep_i\ \parallel\ Activator_i\ \parallel\ (Poll_i\ \parallel\ \Pi_j Prop_{i,j})\ \parallel\ Release_i))\ \backslash\ F\ ]_U$$

where $F = \{done_{i,j} \mid i, j = 1 \ldots n\} \cup \{dispatch_i, done_i, csat_i, ready_i, fin_i \mid i = 1 \ldots n\}$, $U = \{cpu_i | i = 1 \ldots n\}$.

It is straightforward to verify that the above process satisfies the conditions of Theorem 1.1, and, further, I_2 contains all processes $Activator_i$, with $\alpha_i = (dispatch_i, pr_i)$. Consequently, by the same theorem, if a deadlock arises in $RTSystem$, the event $dispatch_i$ is enabled in some process $Activator_i$ but not in the respective $Sleep_i$ process. This implies that the task has not yet completed execution of its previous activation and has missed its deadline. Thus, the system is not schedulable. This completes the proof. □

Thus, the model we have described can be instantiated to a specific task set with a specific scheduling algorithm, and its schedulability can be decided by searching for deadlocks in the resulting model.

4 Conclusions

In this chapter, we have given an overview of how timing information can be embedded in process-algebraic frameworks. We have concentrated on illustrating how time and time passage are modeled in different process algebras, and how the resulting mechanisms interact with each other and with standard, untimed operators. We have proceeded to review the ACSR process algebra which extends the ideas of timed process algebras to the field of timed, resource-constrained systems. We have observed that ACSR shares a number of features with other timed process algebras. For example, all the mentioned formalisms share the view that a timed system is the composition of cooperating sequential processes which synchronize on a shared global clock and operate in a sequence of two-phase steps alternating between time progress and instantaneous steps. Further, in all the formalisms the designer is called to ensure that non-Zenoness and progress are preserved in the system, with respect to the idiosyncracies of the formalism.

ACSR stands out from the remainder of the process algebras considered in that it is geared towards the modeling of resource-sensitive, timed systems. In doing this, it adopts the notion of a *resource* as a first-class entity, and it replaces maximal progress, employed by other process algebras, by the notion of *resource-constrained* progress. Thus, it associates resource-usage with time passage, and it implements appropriate rules to ensure that progress in the system is enforced as far as possible while simultaneous usage of a resource by distinct processes is excluded. In addition, ACSR employs the notions of priorities to arbitrate access to resources by competing processes.

Furthermore, we have illustrated the use of ACSR for the schedulability analysis of a realistic real-time system problem. The systems in question are composed of a set of periodic tasks with precedence constraints, competing for the use of a set of processors, access to which is resolved by a scheduler according to the task priorities. Schedulability analysis of this, and other schedulability problems, is translated in the formalism into the absence of deadlocks in the ACSR process modeling the system. We have made this concrete by providing a compositional result that characterizes the source of deadlocks in a specific set of concurrent components. We have shown that, in our model, this source is associated with tasks missing their deadlines.

ACSR has been extended into a family of process algebras. Extensions and variations include GCSR [4] which allows the visual representation of ACSR processes, Dense-time ACSR [6] which includes a more

general notion of time, ACSR-VP [10] which includes a value-passing capability, PACSR [19], a probabilistic formalism for quantitative reasoning about fault-tolerant properties of resource-bound systems and P^2ACSR [24] which allows to specify power-constrained systems. The PARAGON toolset [22] provides tool support for system modeling and analysis using these formalisms.

References

[1] J. Baeten and C. Middelburg. Process algebra with timing: Real time and discrete time. In *Handbook of Process Algebra*, pages 627–684. Elsevier, 2001.

[2] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):481–529, 1991.

[3] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[4] H. Ben-Abdallah. *GCSR: A Graphical Language for the Specification, Refinement and Analysis of Real-Time Systems*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1996.

[5] H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. S. Kim, I. Lee, and H.-L. Xie. A process algebraic approach to the schedulability analysis of real-time systems. *Real-Time Systems*, 15(3):189–219, 1998.

[6] P. Brémond-Grégoire and I. Lee. Process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189:179–219, 1997.

[7] R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *Proceedings of CONCUR'97*, LNCS 1243, pages 166–180, 1997.

[8] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[10] H. Kwak, I. Lee, A. Philippou, J. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *Proceedings of RTSS'98*, pages 409–419, 1998.

[11] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, pages 158–171, 1994.

[12] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[13] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[14] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of CONCUR'90*, LNCS 458, pages 401–415, 1990.

[15] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proceedings of CAV'91*, LNCS 575, pages 376–398, 1991.

[16] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.

[17] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of* 5^{th} *GI Conference*, LNCS 104, pages 167–183, 1981.

[18] D. Peng, K. Shin, and T. Abdelzaher. Assignment and scheduling of communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12), 1997.

[19] A. Philippou, O. Sokolsky, R. Cleaveland, I. Lee, and S. Smolka. Probabilistic resource failure in real-time process algebra. In *Proceedings of CONCUR'98*, LNCS 1466, pages 389–404, 1998.

[20] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, 1995.

[21] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[22] O. Sokolsky, I. Lee, and H. Ben-Abdallah. Specification and analysis of real-time systems with PARAGON. *Annals of Software Engineering*, 7:211–234, 1999.

[23] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *Proceedings of WP-DRTS'06*, 2006.

[24] O. Sokolsky, A. Philippou, I. Lee, and K. Christou. Modeling and analysis of power-aware systems. In *Proceedings of TACAS'03*, LNCS 2619, pages 409–425, 2003.

[25] W. Yi. Real time behaviour of asynchronous agents. In *Proceedings of CONCUR'90*, LNCS 458, pages 502–520, 1990.

Index