UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
SCIENCES ET GÉOGRAPHIE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Discipline : INFORMATIQUE

présentée et soutenue publiquement par

Gregor Gössler

le 17 septembre 2001

Modélisation Compositionnelle des Systèmes Temps-Réel — Théorie et Pratique

Membres du jury :

Président	Jacques Voiron	Université Joseph Fourier
Rapporteurs	Insup Lee	University of Pennsylvania
	Antoine Petit	ENS de Cachan
Examinateurs	Gérard Berry	Ecole des Mines de Paris
	Pravin Varaiya	UC Berkeley
Directeur de thèse	Joseph Sifakis	CNRS

Compositional Modelling of Real-Time Systems — Theory and Practice

Gregor Gössler

Remerciements

Je remercie Joseph Sifakis, mon directeur de thèse, de m'avoir accueilli à VERIMAG, pour sa confiance, son soutien pendant toute ma thèse et pour avoir été disponible malgré sa grande charge de travail.

Je voudrais remercier les membres de mon jury, Insup Lee et Antoine Petit qui ont accepté d'être rapporteurs, Gérard Berry, Pravin Varaiya et Jacques Voiron.

Merci à Karine, ma collègue de bureau, pour sa bonne humeur et d'innombrables discussions très enrichissantes, scientifiques ou non et pour avoir relu attentivement mon manuscrit.

Je tiens à remercier ...

Oded Maler, Sergio Yovine et Stavros pour leurs encouragements et pour de nombreuses discussions très intéressantes. J'ai eu beaucoup de plaisir à travailler avec eux.

Seb pour nos nombreuses discussions techniques, son "œuf de coucou" et pour son savoir-cuisiner.

Marius et Chaker du "service après-vente" souriant et patient pour leurs outils.

Laurent Mounier, Jacques Briat et Yassine Lakhnech, mes tuteurs et collègues pendant les trois années de monitorat, pour leur confiance et notre collaboration étroite.

Bertrand pour ses soirées légendaires et pour ne m'avoir pas piqué le gâteau au chocolat tout le temps.

Christine, Céline et Katri pour avoir réglé les tâches administratives avec efficacité et gentillesse.

Tous les membres du laboratoire que je n'ai pas cités ici, mais qui m'ont offert un cadre de travail particulièrement fructueux et agréable.

Un grand merci à Fanie et Hélène. C'est les randonnées avec elles qui m'ont donné l'énergie et le moral tellement nécessaires pour la rédaction !

Merci à mes parents et à Viviane pour leur soutien.

Contents

Chapter 1

Introduction

1.1 Modelling Real-Time Systems

Real-time systems, that is, systems whose correctness depends on the respect of timing constraints, have become ubiquitous by now. Nevertheless, their design still poses challenging problems and is a very active domain of research. Real-time systems have to reconcile functional, material, and timing requirements that are often antinomic.

Reactive real-time systems continuously interact with an *environment*. *Embedded* real-time systems are reactive systems residing in a host system that communicates with its environment via physical devices such as sensors, actuators, or a network. Resources are often limited due to the need to reduce the size and cost, as for example in GSM phones. In *hard* real-time systems, not in time means wrong. Even if by far not all hard real-time systems are safety-critical such as automotive control systems [Tri01], their correct functioning is crucial for system availability [Ree97]. Guaranteeing correctness is therefore an essential aspect of real-time systems design.

Reactive real-time systems are in general composed of concurrent, interacting subsystems, or *processes*, which may cooperate, synchronize, and share resources to provide the required service, as specified by *functional requirements*. This coordination is ensured by appropriate synchronization mechanisms.

Moreover, the processes are subject to non-functional requirements, in particular *timing requirements* such as process deadlines, and bounds on response times to events of their environment. The specified timing requirements have to be met in spite of the synchronization constraints, which may require a process to wait for some unavailable resource, or for some other process to complete its computation.

It is the role of a *scheduler* to coordinate the execution of system activities in order to guarantee a correct functioning of the system. The scheduler resolves non-determinism due to competing access of processes to shared resources, restricting the behaviour of the system so as to avoid deadlocks, and to guarantee the timing requirements to be met. The role of the scheduler consists precisely in observing the state of the application and maintaining the requirements satisfied in spite of events of the environment, and of internal actions of the application, such as process arrival or process termination. Guaranteeing correctness of the scheduler is essential for the development of dependable real-time systems.

Modelling is standard engineering practice to deal with the complexity of systems, and understand their behaviour. Modelling a real-time system should allow to validate its design before implementing the system, and to prove its correctness using formal methods. For reactive real-time systems, it is important to build models that faithfully represent their behaviour. In such models, the application has to be modelled together with the behaviour of its environment and its dynamics.

A modelling framework accompanying the design process of real-time systems and providing a methodology, can guide and accelerate the design process, replace ad-hoc solutions by standard constructions, and improve quality of the model. For the framework to be useful, it should meet the following requirements:

- be sufficiently *general* to allow, in a natural and comprehensive way, specification of resource contention, synchronization, priority choice, urgency, preemption, periodic and aperiodic processes, various scheduling disciplines on uni- or multi-processor systems;
- be based, in spite of their expressiveness, on an *analysable and executable* model, that is, be operational rather than descriptive, so as to reduce risks of errors caused by passing from one formalism to another;
- be founded on *theoretical results* ensuring well-defined semantics, supporting a modular specification, compositionality — that is, the semantics of a system can be constructed from the semantics of its components —, and allowing, to some extent, correctness by construction;
- be *practical and applicable*, that is, provide an intuitive high-level modelling formalism, together with a design methodology, and guidelines or standard constructions for common problems. Moreover, it should allow feasible algorithms for automatic analysis, supporting the design process, and be supported by *tools*; and
- help *detecting design errors* by providing *diagnostics* at an early stage, allowing to debug the design, or gain confidence in its correctness, and support a *predictable* model, in the sense that unexpected interaction between separately modelled behavioural requirements is ruled out as far as possible.

Existing formalisms and tools are designed to meet different subsets of the requirements above. However, as some of the items seem difficult to reconcile — for example, generality of the model, and support for early detection of design errors —, they are not equally addressed by one framework. The following section gives an overview of related work.

1.1.1 Prior Research

Real-time scheduling. Well established theory and scheduling algorithms have been successfully applied to real-time systems development. [SSNB95, GMR95, GRS96, BLMSV98] give an overview of the large collection of results for real-time scheduling. These results essentially consist of a hypothesis under which the result applies, a scheduling policy specifying how and when a resource is granted to a waiting process, and a sufficient condition for the scheduling to be feasible, that is, for the timing requirements to be guaranteed.

Available results are applicable when the processes and their timing constraints meet specific hypotheses about periods, execution times, deadlines, resource allocation, processor utilization etc. In this case, criteria are provided under which schedulability — that is, respect of the timing constraints — is guaranteed for particular scheduling policies such as rate-monotonic scheduling (RMS) [LL73, LSD89], earliest-deadline-first scheduling (EDF) [LL73], and deadline-monotonic scheduling (DMS) [ABR91].

Schedulability analysis essentially consists in checking that the system meets the schedulability criteria prescribed by the theory, which allows efficient schedulability analysis tools. It does not require the use of a model representing the dynamic behaviour of the system to be scheduled. Current engineering practice essentially adopts this approach, see for example [HKO$^+$93].

Existing scheduling theory requires the application to fit into the mathematical framework of the schedulability criterion. Studies to relax such hypotheses have been carried out, for example, [Bal98] proposes an algorithm assigning fixed priorities to a system of processes that can trigger one another, in case they are schedulable with fixed priorities. [Ves94] provides an extension of rate monotonic analysis taking blocking on non-preemptable resources into account, and allowing quantitative results about laxity, that is, the factor by which the computation time of processes can be increased without compromising their schedulability. HRT-HOOD [BW95] is a method for designing real-time systems in ADA in such a way that the model fits into the theory of fixed-priority scheduling, and can be analysed for schedulability. Similarly, [SK00] integrates schedulability analysis with object-oriented modelling.

However, most of these schedulability results apply only for particular process models or do not allow complex interaction between the components such as shared resources apart from the processor, atomicity, or communication. Generally, functional and timing properties are specified and verified separately, and no unified approach for general scheduling problems has been proposed so far.

Model-based approaches. To overcome these limitations, an alternative approach consists in building explicitly a timed model of the real-time application, that is, the application processes together with their possible interaction, and verifying schedulability [BPS00] or extracting a scheduler [NY00], without considering particular scheduling policies.

The controller synthesis paradigm for discrete-event systems [RW87] and timed systems [AMP95, MPS95, Alt98, TA99] provides a general framework for scheduling. This is the most general approach, but the algorithmic method for synthesizing a controller is of prohibitive complexity. For this reason, sometimes, the existence of an invariant implying satisfaction of the timing constraints is explored, using real-time verification techniques [HNSY94, ACH+95, BSS97, KLP+98, Tri98, Pet99] and tools such as KRONOS [DOTY96, Yov97], UPPAAL [LPY97, JLS00], VERUS [CCMM95], COSPAN [HHK96], or HYTECH [HHWT97]. A non-empty invariant satisfying the timing constraints is a sufficient condition for schedulability, requiring techniques of lower complexity than synthesis which do not distinguish between controllable and uncontrollable actions.

There are several other approaches to tackle the complexity of verifying real-time systems, or synthesizing schedulers. For example, [BV95] discusses incremental verification of communicating Time Petri Nets, based on assume-guarantee reasoning. [MTdR96] presents a scheduler synthesis tool based on constraint satisfaction for a simple process model that nevertheless allows shared resources, and a timing specification in Real-Time Logic. [BG98] discusses the analysis of non-deterministic real-time systems using the $(\max, +)$ algebra, which does not require exploring the state space like traditional model-checking techniques. [Die99] provides an algorithm synthesizing a programmable logic controller from a specification described by a fragment of the duration calculus.

Modelling methodologies and tools for real-time systems have shifted into the focus of research in the recent years. [Lee98] describes a formal low-level framework for real-time system models, where processes are described by sets of possible behaviours. This framework is intended as a unifying meta-model rather than to directly model real-time applications. [Ves99] discusses modelling and verification of preemptive real-time systems with hybrid automata. Similarly, [Cor96b] describes a methodology for modelling a general class of real-time systems with resource constraints, synchronization and context switching overhead, and atomicity of code segments, as hybrid systems. The method is applied to the timing analysis of Ada programs. Adopting the same framework, [ACD98] discusses the timing analysis of partially implemented systems, where lacking pieces of code are specified in Graphical Interval Logic. [LA99, LAM99] discuss a formal model of the Ravenscar [DB98] subset of Ada 95, allowing to verify applications using the model-checker UPPAAL.

Among the modelling and design tools, we shall mention the PTOLEMY [L+01] project and toolset aiming at heterogeneous modelling, simulation, and design of embedded systems by integrating different models of computation. Another tool for the integration of heterogeneous models is the SPI Workbench [EZR+99], which uses graphs of communicating processes annotated with timing intervals, as a unifying abstract representation serving as a basis for verification and hardware/software co-design. GIOTTO [HHK01] is a tool-supported design methodology for distributed embedded systems based on the time-triggered paradigm. It consists of a programming language, and a platform-dependent part including a compiler and a runtime library. TAXYS [BPS00, CPP+01] is a tool for the development and verification of embedded systems in the telecommunication domain. The system and its environment are specified in the synchronous language ESTEREL [BG92] annotated with timing constraints. The model can be verified by the model-checker KRONOS, and compiled to C code by the ESTEREL compiler SAXO-RT [WBC+00].

There is some work aiming at integrating model-based analysis of real-time systems, and scheduling theory. The interest of considering particular scheduling policies in a model-based approach is twofold. First, it allows to verify both the functional correctness, and the timeliness, of a scheduled real-time system, whereas the same system without a scheduler, generally does not meet its timing constraints. Second, restricting the set of possible behaviours helps to manage the state explosion problem. Most of this work is based on process algebras extended with a notion of priority.

[BCL99] defines a process algebra based on CCS [Mil89] with real-time semantics and dynamic priorities. In the process algebra RTSL (Real-Time Specification Language) [FC93], scheduling policies such as RMS and EDF can be modelled by a function associating, with any system state, a subset of processes that remain enabled after priority choice.

The process algebra ACSR (Algebra of Communicating Shared Resources) [LBGG94, BGL97] provides a framework with discrete and dense-time semantics for modelling coordination between processes including shared resources, synchronization, preemption, static priorities, and exception handling. A prioritized strong bisimulation ensures compositionality. The PARAGON toolset [BALS99] for the specification and verification of real-time systems is based on ACSR. The system can be modelled in a graphical specification language. Verification is done by state-space exploration, or checking for bisimulation with a process specifying a high-level behaviour. [KLP+98] discusses the modelling of real-time schedulers in ACSR-VP, an extension of ACSR with value passing communication. Schedulability analysis amounts to symbolically checking the — possibly parameterized – model for bisimulation with a non-blocking process, and synthesizes the parameter values for which the system is schedulable. In [BACC+98], models of basic process specifications are given, and schedulers for EDF and the priority inheritance protocol [SRL90] are modelled under ACSR-VP.

[JJ01] presents a modelling methodology for fault-tolerant distributed real-time systems. Processes and fault models are specified in a process algebra based on Timed CCS; liveness properties and deadlines are expressed in logic based on modal timed μ-calculus. The authors give examples of a best-effort EDF scheduler, and a planning-based scheduler where processes are only scheduled if their deadlines are guaranteed to be met.

[EWY99] model real-time processes scheduled under EDF as timed automata, and model-check the obtained representation using UPPAAL. However, their modelling method is not compositional. [Bra00] introduces I/O timed components, essentially timed automata with an interface declaration, as a modelling formalism guaranteeing non-zeno and non-blocking synchronization by construction. Information about the interface of I/O timed components is used by a *relevance calculus* to make abstraction from components that are irrelevant for proving a given property specified as an observer process.

MetaH [Ves97] is a development tool initially designed for avionics applications. It accompanies the development process of real-time systems from the specification down to code generation, and implements schedulability analysis based on the results of [Ves94, Bin97] extending rate-monotonic analysis. It is also possible to specify error models, and carry out reliability analysis.

Sometimes a deductive approach is used to verify correctness of a scheduler [FKU95, Dut00, DS00] using theorem provers. In [LJ99] and [LJ01], real-time programs with timing constraints, fault models, and scheduling policies are modelled in the logic TLA [Lam94]. Proving that scheduling the real-time system under a certain discipline, both specified in TLA, is feasible, amounts to verifying a schedulability condition similar to the results from scheduling theory.

1.2 Approach and Contribution of this Thesis

To model a real-time application, we adopt the architecture shown in figure 1.1, which suggests a decomposition and an incremental description principle. The real-time application model is built from asynchronous timed systems representing the processes, by successive application of a *synchronization layer* and of a scheduler.

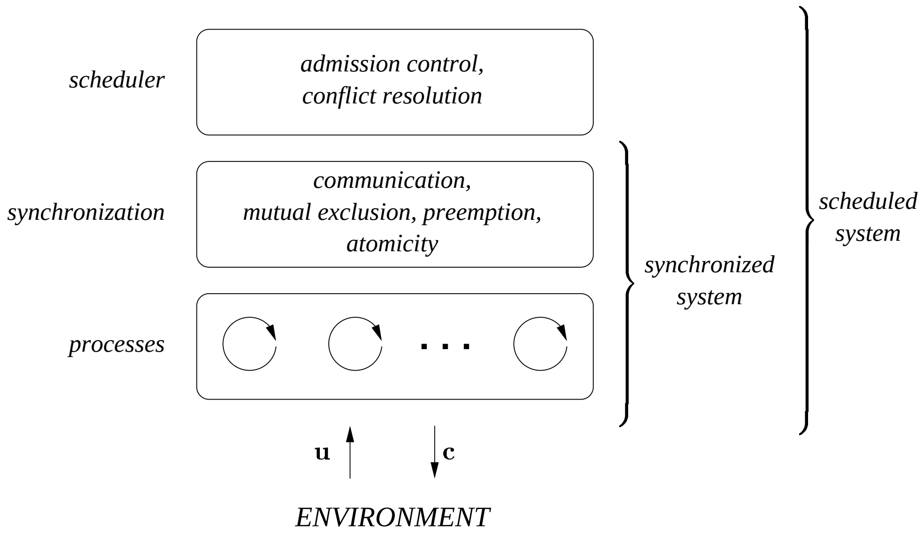


Figure 1.1: Layered architecture of a real-time application.

Processes are modelled as timed systems, a variant of timed automata with urgency and the possibility to stop clocks. The general framework of timed systems allows the specification of complex process models. The timed systems carry all timing information of the processes, such as execution times, deadlines, and end-to-end timing constraints. Moreover, for modelling reactive systems interacting with an environment, the processes can include an abstract model of the environment, such as the arrival of sporadic events with assertions about minimal and maximal inter-arrival times. Events of the environment, and of internal actions of the application representing nondeterminism of the processes — such as process arrival or process termination

— are represented by *uncontrollable* actions, whereas actions giving access to shared resources are *controllable* and can be disabled so as restrict the behaviour of the processes to guarantee the functional and timing requirements of the system. In the figure, **u** stands for uncontrollable, **c** stands for controllable actions.

The synchronization layer ensures inter-process communication, mutual exclusion on shared resources, and atomicity of action sequences, and provides a preemption mechanism. It is in charge of guaranteeing only functional properties of the application. The processes coordinated by the synchronization layer, are referred to as the *synchronized system*.

The scheduler interacts with the underlying system by resolving non-determinism so as to satisfy non-functional properties, timing requirements in particular. It consists of admission control rules deciding whether some process is eligible for resource allocation, and a scheduling policy resolving conflicts between processes waiting for the same resource. The whole system, consisting of the synchronized system and the scheduler, forms the *scheduled system*.

In this modelling framework, behavioural restrictions and coordination such as mutual exclusion, the resource allocation policy, and the process dispatching mechanism are specified separately as priority functions, and composed to obtain the scheduled system. We believe that priority functions are an intuitive, yet powerful means to dynamically restrict the system behaviour. They are instrumental for the specification of both the synchronization layer and the scheduler in our model-based framework, creating a general unified modelling framework for real-time systems. We illustrate the elegance of priority functions with several examples of well-known scheduling policies. Composability of priority functions allows a modular specification of coordination between processes, and detection of inconsistency between the different behavioural aspects.

Compositional semantics is a prerequisite for developing modular reasoning and design principles. We define structural properties for well-timedness, liveness, and safety of timed systems, which are inexpensive to verify, and provide preservation results allowing compositional reasoning about these properties. Based on these results, a priority synthesis algorithm is presented which constructs, if possible, a safe and live system from a partial or inconsistent specification.

Most of the results presented in this thesis have been implemented in the compositional modelling tool PROMETHEUS, which has been designed to support the design of real-time systems. Particular attention has been paid to hide technical details from the user, allowing to specify processes, synchronization, and the scheduler in a high-level description language. In order to give access to existing verification tools, PROMETHEUS has been integrated in a validation platform developed at VERIMAG.

1.2.1 Organization of this Document

Chapter 2 introduces timed systems with priorities as the modelling framework used throughout this thesis. It concludes with some reflections about modelling time. In chapter 3, guidelines for modelling real-time processes and the synchronization layer are given. Chapter 4 defines structural liveness and safety properties, and presents results for compositional reasoning about these properties. Chapter 5 discusses the modelling of real-time schedulers in our framework, and gives examples for several well-known scheduling policies. It conclusdes with a sketch of a model-based timing analysis algorithm based on priority functions. In chapter 6, the modelling tool PROMETHEUS is presented, and its utilization is explained. Chapter 7 applies the results in several case studies. In Chapter 8, we conclude and have a look at future work.

Chapter 2

Timed Systems with Priorities

In order to model real-time applications, we use timed systems whose transitions are submitted to timing constraints expressed as predicates on variables called clocks. Priorities are associated with a timed system to restrict its behaviour.

Preliminaries

Before coming to timed systems, we will define some frequently used notations.

Let $X = \{x_1, \ldots, x_m\}$ be a set of variables called *clocks*, defined on the domain $\mathbb{D}$. For a continuous-time interpretation, take $\mathbb{D} = \mathbb{R}_{\geqslant 0}$, the set of non-negative reals, and for a discrete-time interpretation, take $\mathbb{D} = \mathbb{N} = \{0, 1, 2, \ldots\}$. Let $\mathbb{D}_{>0} = \mathbb{D} \setminus \{0\}$. A vector $\mathbf{x} \in \mathbb{D}^m$ is called *clock valuation.* A predicate on $\mathbb{R}^m_{\geqslant 0}$ is called *X-constraint.* X-constraints are defined on $\mathbb{R}^m_{\geqslant 0}$ so that they can be interpreted at the points in $\mathbb{D}^m$ for both $\mathbb{D} = \mathbb{R}_{\geqslant 0}$ and $\mathbb{D} = \mathbb{N}$.

Definition 2.1 ($\Diamond$ and $\diamondminus$) *For a given X-constraint C, $k \in \mathbb{N} \cup \{\infty\}$, and a rate vector $\mathbf{b} \in \{0,1\}^m$, the X-constraint $\Diamond^{\mathbf{b}}_k C$ ("eventually C within k with respect to $\mathbf{b}$") is defined as $\forall \mathbf{x} \in \mathbb{R}^m_{\geqslant 0}$,*

$$(\Diamond^{\mathbf{b}}_k C)(\mathbf{x}) = \exists d \leqslant k \,.\, \mathbf{x} + d\mathbf{b} \in \mathbb{D}^m \ \wedge\ C(\mathbf{x} + d\mathbf{b}) \ ,$$

where $\mathbf{x} + d\mathbf{b}$ is the clock valuation obtained from $\mathbf{x}$ by increasing by d the clock values for which the component of $\mathbf{b}$ is equal to one. We write $\Diamond^{\mathbf{b}}$ for $\Diamond^{\mathbf{b}}_\infty$.

The X-constraint $\diamondminus^{\mathbf{b}} C$ ("once in the past C with respect to $\mathbf{b}$") is defined as $\forall \mathbf{x} \in \mathbb{R}^m_{\geqslant 0}$,

$$(\diamondminus^{\mathbf{b}} C)(\mathbf{x}) = \exists \mathbf{x}' \in \mathbb{D}^m \ \exists d \in \mathbb{D} \,.\, C(\mathbf{x}') \ \wedge\ \mathbf{x}' + d\mathbf{b} = \mathbf{x} \ .$$

For both operators, $\mathbf{b}$ may be omitted if $\mathbf{b}[x] = 1$ for all clocks $x \in X$.

The operators $\Diamond$ and $\diamondminus$ are just a notation for existential quantification over time and should not be confused with temporal logic operators.

Example 2.1 *For* $X = \{x\}$, $C = (2 < x \leqslant 5)$, *and* $\mathbf{b}[x] = 1$, *we have*

$$
\begin{array}{rcl}
\Diamond_0^{\mathbf{b}}(C) & = & 2 < x \leqslant 5 \ = \ C \\
\Diamond_1^{\mathbf{b}}(C) & = & 1 < x \leqslant 5 \\
\Diamond_2^{\mathbf{b}}(C) & = & 0 < x \leqslant 5 \\
\Diamond_k^{\mathbf{b}}(C) & = & 0 \leqslant x \leqslant 5 \quad \textit{for } k \geqslant 3, \textit{ and} \\
\diamondminus^{\mathbf{b}}(C) & = & x > 2 \,.
\end{array}
$$

For $\mathbf{b}[x] = 0$ *and* $k \in \mathbb{N} \cup \{\infty\}$, $\Diamond_k^{\mathbf{b}}(C) = \diamondminus^{\mathbf{b}}(C) = C$.

For two clocks $X' = \{x, y\}$, $C' = (2 < x \leqslant 5 \ \wedge \ 3 \leqslant y \leqslant 4)$, *and rate vectors* $\mathbf{b}_1$ *and* $\mathbf{b}_2$ *with* $\mathbf{b}_1[x] = \mathbf{b}_1[y] = 1$, *and* $\mathbf{b}_2[x] = 1$, $\mathbf{b}_2[y] = 0$, *we have*

$$
\begin{array}{rcl}
\Diamond^{\mathbf{b}_1}(C') & = & x \leqslant 5 \ \wedge \ y \leqslant 4 \ \wedge \ -2 < x - y \leqslant 2 \\
\Diamond^{\mathbf{b}_2}(C') & = & x \leqslant 5 \ \wedge \ 3 \leqslant y \leqslant 4 \\
\diamondminus^{\mathbf{b}_1}(C') & = & 2 < x \ \wedge \ 3 \leqslant y \ \wedge \ -2 < x - y \leqslant 2 \\
\diamondminus^{\mathbf{b}_2}(C') & = & 2 < x \ \wedge \ 2 \leqslant y \leqslant 4 \,.
\end{array}
$$

Figure 2.1 shows the X*-constraints* $\Diamond^{\mathbf{b}_1}(C')$, $\Diamond^{\mathbf{b}_2}(C')$, $\diamondminus^{\mathbf{b}_1}(C')$, *and* $\diamondminus^{\mathbf{b}_2}(C')$.

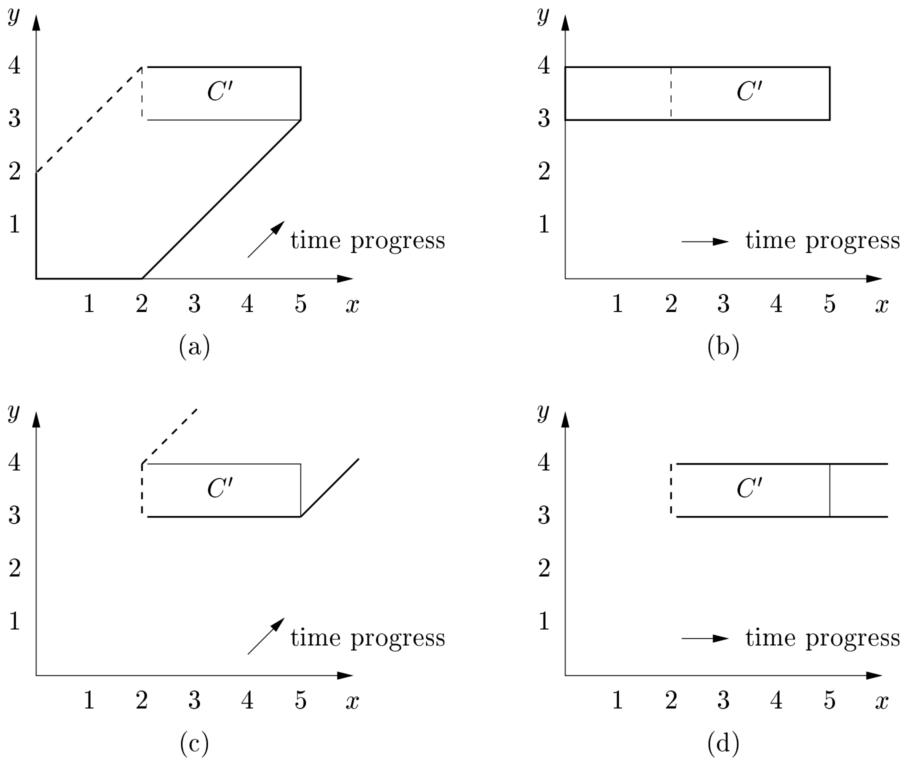


Figure 2.1: (a) $\Diamond^{\mathbf{b}_1}(C')$; (b) $\Diamond^{\mathbf{b}_2}(C')$; (c) $\diamondminus^{\mathbf{b}_1}(C')$; (d) $\diamondminus^{\mathbf{b}_2}(C')$ for $\mathbf{b}_1 = (1, 1)$ and $\mathbf{b}_2 = (1, 0)$.

Property 2.1 *For any X-constraints C, C', rate vector $\mathbf{b}$, and $k, l \in \mathbb{N} \cup \{\infty\}$,*

- $\Diamond_k^{\mathbf{b}}(C \vee C') = \Diamond_k^{\mathbf{b}} C \ \vee \ \Diamond_k^{\mathbf{b}} C'$ *(distributivity with respect to the disjunction);*
- $\Diamond_k^{\mathbf{b}}(\Diamond_l^{\mathbf{b}} C) = \Diamond_{k+l}^{\mathbf{b}} C$ *(additivity);*
- *If $C' \implies C$, then $\Diamond_k^{\mathbf{b}} C' \implies \Diamond_k^{\mathbf{b}} C$ (monotony);*
- *If $l \geqslant k$, then $\Diamond_k^{\mathbf{b}} C \implies \Diamond_l^{\mathbf{b}} C$ (weakening).*

2.1 Timed Systems

In order to model reactive real-time systems, we use timed systems with two kinds of actions: *controllable* actions that can be seen as internal or output actions, and *uncontrollable* actions that model inputs from the environment. Timed systems with priorities, but without the possibility of stopping clocks, have been introduced and studied in [BS97, BS98, BS00, BGS00] as a variant of timed automata [AD94]. Timed systems are an abstraction of timed automata with deadlines [BST98, BS00] with clocks that can be stopped, and thus a particular case of hybrid systems [KPSY92, ACH+95].

Definition 2.2 (Timed system) *A* timed system *is a tuple $(S, A, T, X, \mathbf{b}, h)$ where*

- *S is a finite set of* control states*;*
- *A is a finite vocabulary of* actions*, partitioned into two sets of controllable actions A^c and uncontrollable actions A^u;*
- *$T \subseteq S \times A \times S$ is an untimed* transition relation*;*
- *$X = \{x_1, \ldots, x_m\}$ is a set of clocks;*
- *$\mathbf{b} : S \to \{0,1\}^m$ is a function mapping each state to a boolean* rate vector*;*
- *h is a function labelling each untimed transition $(s, a, s') \in T$ with a tuple $h(s, a, s') = (g, \tau, r)$, where g is an X-constraint called* guard*, $\tau \in \{\lambda, \delta, \epsilon\}$ is an* urgency type*, respectively* lazy*,* delayable*, and* eager*, and $r \subseteq X$ is a set of clocks to be* reset*.*

For $t = (s, a, s') \in T$, we call $(s, a, h(t), s')$ a *timed transition*. The definition of guards as general X-constraints relaxes usual syntactical restrictions ensuring decidability.

We write $\mathbf{b}_s$ for $\mathbf{b}(s)$. In figures, we omit $\mathbf{b}_s$ if $\mathbf{b}_s[x] = 1$ for all $x \in X$, and we omit the set r for transitions with $r = \emptyset$. We extend the notion of controllability from an action to the transitions labelled by that action. For $s \in S$, let $T(s) \subseteq T$ denote the set of all untimed transitions issued from s. For any $T' \subseteq T$, let $\mathit{actions}(T') = \{a \mid \exists s, s' \in S \,.\, (s, a, s') \in T'\}$ be the set of actions labelling the transitions in T'.

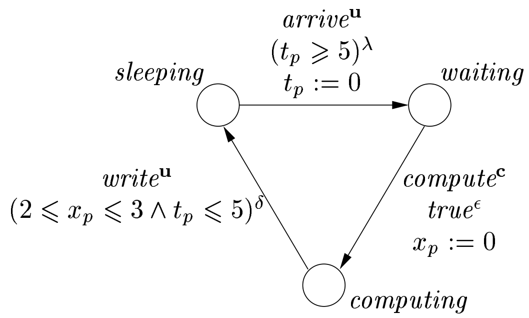


Figure 2.2: Timed system P modelling a cyclic process.

Example 2.2 *Figure 2.2 shows the timed system* P *modelling a cyclic process. From state* sleeping*, the uncontrollable transition* arrive *leading to state* waiting *may be taken as soon as the value of clock t_p is at least 5. t_p is reset by* arrive*; it is used to measure the time elapsed since the last occurrence of* arrive*. Hence, the process has a minimal inter-arrival time of 5. The controllable transition* compute *leading from* waiting *to* computing *is always enabled and resets another clock x_p measuring the time passed in state* computing*. Finally, with the uncontrollable transition* write*, the process returns to state* sleeping *after having spent between 2 and 3 time units in* computing*, and at most 5 time units after its last arrival.*

Definition 2.3 (Semantics of a timed system) *A* state *of a timed system $TS = (S, A, T, X, \mathbf{b}, h)$ is a pair $(s, \mathbf{x}) \in S \times \mathbb{D}^m$, where $\mathbf{x}$ is a clock valuation. TS defines a transition relation $\rightarrow^{TS} \subseteq (S \times \mathbb{D}^m) \times (A \cup \mathbb{D}) \times (S \times \mathbb{D}^m)$ as follows. For any state $(s, \mathbf{x}) \in S \times \mathbb{D}^m$,*

- $(s, \mathbf{x}) \xrightarrow{a}{}^{TS} (s', \mathbf{x}[r])$ *if* $(s, a, s') \in T$ *with* $h(s, a, s') = (g, \tau, r)$ *and* $g(\mathbf{x}) = true$. $\mathbf{x}[r]$ *is the clock valuation obtained from* $\mathbf{x}$ *by resetting all clocks in r to zero, and leaving the others unchanged.*

- *Let $h(t_i) = (g_i, \tau_i, r_i)$ for all $t_i \in T(s)$. For any delay $d \in \mathbb{D}$,*

$$(s, \mathbf{x}) \xrightarrow{d}{}^{TS} (s, \mathbf{x} + d\mathbf{b}_s) \iff \bigwedge_{t_i \in T(s)} \begin{cases} \neg(\Diamond^{\mathbf{b}_s} g_i)(\mathbf{x}) \ \vee \ (\Diamond^{\mathbf{b}_s} g_i)(\mathbf{x} + d\mathbf{b}_s) & \textit{if } \tau_i = \delta \\ \forall d' \in \mathbb{D},\ d' < d \,.\, \neg g_i(\mathbf{x} + d'\mathbf{b}_s) & \textit{if } \tau_i = \epsilon \\ \textit{true} & \textit{if } \tau_i = \lambda \,. \end{cases}$$

Transitions labelled with elements of A are *discrete transitions*, whereas transitions labelled with elements of $\mathbb{D}$ are *time steps*. The definition above means that a discrete transition is enabled and can be taken when its guard is *true* for the current clock valuation. The transition is instantaneous, and the clocks in the reset set associated with the transition are reset to zero. In any control state, time can progress until a transition leaving the control state becomes *urgent*. A lazy transition is never urgent and may never be taken; a delayable transition can be delayed as long as time progress does not disable it forever; and an eager transition is urgent whenever it is enabled.

When the context is clear, we write $\rightarrow$ instead of $\rightarrow^{TS}$.

Example 2.3 *The transition* arrive *of the process in example 2.2 is lazy, which means that the process may stay in* sleeping *forever. Transition* compute *is always enabled and eager, which means that it must be taken immediately after entering the state* waiting. *Transition* write *is delayable, specifying that the process may remain in* computing *as long as* $x_p \leqslant 3$ *and* $t_p \leqslant 5$*, but must leave the state before* write *becomes disabled, that is, at last at* $x_p = 3 \wedge t_p \leqslant 5 \ \vee \ 2 \leqslant x_p \leqslant 3 \wedge t_p = 5$.

In order to express coordination between two or more timed systems, two basic concepts are used throughout this thesis: priority choice, and parallel composition with synchronization.

2.2 Priority Choice

Priorities are widely used in modelling formalisms to restrict system behaviour, especially for conflict resolution [BBK86, CW95, CLN98, BGL97, KLP+98, BACC+98, BCL99, JJ01, Phi01] or to reduce non-determinism.

Priorities can be assigned between processes or between actions, and they can be static or dynamic, that is, depend on the system state or the execution. Generally, priorities are specified by assigning *absolute* priority levels to processes or actions, which leads to a *stratified* priority model. Another possibility is to define priorities as a (partial) order on the set of processes or actions. We refer to this second case as *relative* or *unstratified* priority. Table 2.1 shows a non exhaustive overview of existing priority choice formalisms.

		static	dynamic
absolute	process-level	fixed-priority sched.	priority ceiling protocol
	transition-level	SDL	most process algebras
relative	process-level		dynamic-priority scheduling
	transition-level		process algebras [CW95], [Phi01], *priority functions*

Table 2.1: Overview of priority choice in modelling reactive systems.

Priorities between processes are widely used for scheduling, see for example [LL73, LSD89, SRL90, ABR91, HKO+93]. However, assigning priorities on the transition level allows to express more fine-grained properties.

In most approaches allowing priorities between transitions, absolute priority levels are assigned to transitions, and any time there is a conflict between transitions of different priorities, one of the highest-priority transition is taken. In practice, these priorities are either assigned by hand, often expressing only locally a priority relation between the transitions of one process [IT99], or they are assigned dynamically, which is typically the case for process algebras with priorities. However, the former case of statically assigned absolute priorities yields models that are not compositional, in the sense that adding a process in general requires recomputing the priorities. This problem is overcome in the latter case of dynamically assigned absolute priorities. In that case, priorities can be computed depending on the current system state, but the priority order itself is not compositional, in the sense that two priority assignments expressing two properties cannot be easily composed to a single priority assignment ensuring both properties. Moreover, it is less straight-forward to express properties related to partial orders on processes or actions, such as local preemption [CLN01].

For these reasons, relative priorities, that is, a partial priority order between the actions of the processes, are particularly interesting, since they allow to express *local* properties as priority relations that only apply between certain transitions, without side effect on other transitions. Properties can be specified in a modular manner as priority orders, which can then be composed while checking their consistency. Process algebras with unstratified priorities are discussed in [CW95, Phi01]. We adopt and generalize the approach developed in [BS97, BS00, BGS00], providing a general framework for dynamic relative priorities between process actions.

Let $TS = (S, A, T, X, \mathbf{b}, h)$ be a timed system. Assigning priorities between actions when from a state several actions are enabled, amounts to restricting the guards of the actions of lower priority to leave precedence to the actions of higher priority. Consider for example, two transitions $t_i = (s, a_i, s_i)$, $i \in \{1, 2\}$, with a common source state s and $h(t_i) = (g_i, \tau_i, r_i)$. If a_1 has lower priority than a_2, then the guard of t_1 becomes g_1' with $g_1' \implies g_1$, while g_2 remains unchanged. Commonly, g_1' is taken equal to $g_1 \wedge \neg g_2$ which means that when a_1 and a_2 are simultaneously enabled in the system without priorities, only a_2 is enabled in the prioritized system. However, for timed systems it is possible to define priority relations leaving precedence to an action that is known to become enabled within some finite time.

Coming back to the previous example, we can take $g_1' = g_1 \wedge \neg \Diamond_k^{\mathbf{b}_s} g_2$ for some finite k, or even $g_1' = g_1 \wedge \neg \Diamond^{\mathbf{b}_s} g_2$. In the former case a_1 gives priority to a_2 if a_2 is eventually enabled within a delay k. In the latter case, a_1 is disabled if a_2 is not disabled forever.

The following definition of priority order has been introduced and studied in [BS97].

Definition 2.4 (Priority order) *Let $\prec \subseteq A \times (\mathbb{N} \cup \{\infty\}) \times A$ be a relation. We write $a_1 \prec_k a_2$ for $(a_1, k, a_2) \in \prec$. The relation $\prec$ is a* priority order *if for all $a_1, a_2, a_3 \in A$ and for all $k, l \in \mathbb{N} \cup \{\infty\}$,*

- *$\prec_k$ is a strict partial order;*
- *$a_1 \prec_k a_2$ implies $\forall k' \in \mathbb{N} \,.\, k' < k \,.\, a_1 \prec_{k'} a_2$ (monotony);*
- *$a_1 \prec_k a_2$ and $a_2 \prec_l a_3$ implies $a_1 \prec_{k+l} a_3$ (transitivity).*

Property 2.2 *For any priority order $\prec$, $\prec_0 = \bigcup_{k \in \mathbb{N} \cup \{\infty\}} \prec_k$, and $\prec_0$ is a strict partial order.*

An action a_j is said to *dominate* another action a_i in the priority order $\prec$ if $a_i \prec_0 a_j$. a_j is *minimal* in $\prec$ if it does not dominate any other action; a_j is *maximal* if there is no action that dominates a_j in $\prec$.

Hypothesis: In order to ensure that uncontrollable actions cannot be disabled by priority choice, we assume that uncontrollable actions are maximal.

Notation: For two action sets A_1 and A_2 and $k \in \mathbb{N} \cup \{\infty\}$, we write $A_1 \prec_k A_2$ for $\forall (a_1, a_2) \in A_1 \times A_2 \,.\, a_1 \prec_k a_2$.

Definition 2.5 (Restriction) *Let $TS = (S, A, T, X, \mathbf{b}, h)$ be a timed system, $s \in S$, and $\prec$ be a priority order, and suppose that for all $t_i = (s, a_i, s_i) \in T(s)$, $h(t_i) = (g_i, \tau_i, r_i)$. For $t_i \in T(s)$, the* restriction *of g_i according to $\prec$, noted $g_i/\prec$, is defined as*

$$g_i/\prec \;=\; g_i \;\wedge\; \neg \bigvee_{\substack{(s,a_j,s_j)\in T(s)\\ a_i \prec_k a_j}} \Diamond_k^{\mathbf{b}_s} g_j \;.$$

Intuitively, transition t_i becomes disabled under $\prec$ when there is a transition (s, a_j, s_j) labelled with a dominating action a_j with $a_i \prec_k a_j$ for some k that will be enabled within delay k. In other words, the parameter k specifies the delay one is ready to wait to give priority to the dominating action. The special cases $\prec_0$ and $\prec_\infty$ specify *immediate* and *unbounded* priorities, respectively.

Example 2.4 *Consider a state s with two leaving transitions $t_i = (s, a_i, s_i)$ with $h(t_i) = (g_i, \tau_i, r_i)$ for $i \in \{1, 2\}$. Suppose that $g_1 = true$, $g_2 = (2 < x \leqslant 5)$, and $\mathbf{b}_s[x] = 1$. For $\prec = \{a_1 \prec_k a_2\}$ with a given k, we have $g_1/\prec = g_1 \wedge \neg \Diamond_k g_2$. Figure 2.3 shows $g_1/\prec$ for different values of k.*

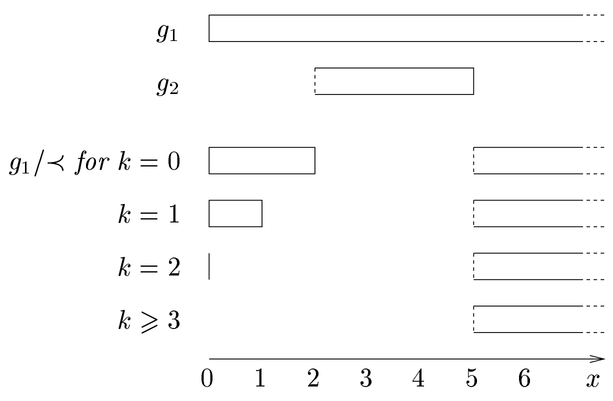


Figure 2.3: $g_1/\prec$ for the possible values of k.

We allow priorities depending on the current system state, that is, the control state and the clock valuation.

Definition 2.6 (State constraint) *A predicate on* $S \times \mathbb{R}^m_{\geqslant 0}$ *is called* state constraint. *For a state constraint* C *and a control state* $s \in S$*, let* $C(s)$ *be the* X*-constraint that* C *associates with* s*, defined as* $\forall \mathbf{x} \in \mathbb{R}^m_{\geqslant 0}$, $\big(C(s)\big)(\mathbf{x}) \iff C\big((s,\mathbf{x})\big)$.

Definition 2.7 (Priority function) *A* priority function pr *is a finite set of pairs* $\{(C^j, \prec^j)\}_{j \in J}$*, where* J *is an index set; for any* $j \in J$*,* $\prec^j$ *is a priority order or* $\perp$ *(read "undefined"), and* C^j *is a state constraint that specifies when the priority order applies, such that* $j_1 \neq j_2 \implies C^{j_1} \wedge C^{j_2} = \mathit{false}$*. The priority order associated with some state* $(s,\mathbf{x}) \in S \times \mathbb{R}^m_{\geqslant 0}$ *is*

$$pr(s,\mathbf{x}) = \begin{cases} \prec^j & \text{if } \exists j \in J \;.\; C^j(s,\mathbf{x}) \\ \emptyset & \text{otherwise} \;. \end{cases}$$

The priority function pr *is called* well-defined *if* $\forall (s,\mathbf{x}) \in S \times \mathbb{D}^m \;.\; pr(s,\mathbf{x}) \neq \perp$.

An action is *minimal* (resp. *maximal*) in pr if for any $(C, \prec) \in pr$, it is minimal (resp. maximal) in $\prec$.

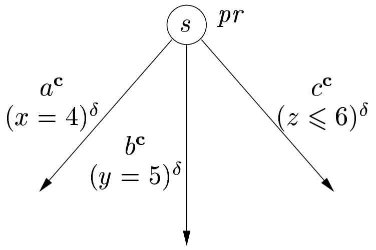
Any timed system with an associated priority function represents a timed system without priorities:

Definition 2.8 (Timed system with priorities) *A* timed system with priorities *is a tuple* (TS, pr)*, where* $TS = (S, A, T, X, \mathbf{b}, h)$ *is a timed system, and* pr *is a priority function.* (TS, pr) *represents a timed system* $TS' = (S, A, T', X, \mathbf{b}, h')$ *with* $T' = \{(s, a, s') \in T \mid \forall \mathbf{x} \in \mathbb{D}^m \;.\; pr(s,\mathbf{x}) \neq \perp\}$*. For all* $t = (s,a,s') \in T'$ *with* $h(t) = (g, \tau, r)$*,* $h'(t) = (g', \tau, r)$*, where*

$$g' \;=\; g \;\wedge \bigwedge_{\substack{(C,\prec) \in pr \\ \prec \neq \perp}} \big(g/\!\prec \;\vee \neg C(s)\big) \;.$$

Intuitively, the guard g of any transition of TS is restricted according to the priority orders $\prec$. This restriction applies only at clock valuations for which $C(s)$ holds, where C is the state constraint associated with $\prec$, and s is the source state of the transition. Control states for which the priority function is undefined for some clock valuation $\mathbf{x}$, are considered as error states with no leaving transition.

Example 2.5 *Consider the three transitions leaving state* s *of a timed system* TS *as in figure 2.4. Applying the priority function* $pr = \big\{\big(s \wedge (y < z), a \prec_\infty b\big),\ \big(s \wedge (y \geqslant z), a \prec_\infty b, c \prec_0 b\big)\big\}$ *to* TS *restricts the guards* $g_a = (x = 4)$*,* $g_b = (y = 5)$*, and* $g_c = (z \leqslant 6)$ *of the three transitions as shown on the right.*



$$\begin{aligned}
g'_a &= g_a \wedge \big(\neg(y < z \vee y \geqslant z) \ \vee\ \neg\Diamond(y=5)\big) \\
&= (x = 4 \ \wedge\ y > 5) \\
g'_b &= g_b \ =\ (y=5) \\
g'_c &= g_c \wedge (\neg(y \geqslant z) \ \vee\ y \neq 5) \\
&= \big(z \leqslant 6 \ \wedge\ (y < z \ \vee\ y \neq 5)\big)
\end{aligned}$$

Figure 2.4: Priority choice in state s.

It is often desirable to restrict a timed system by several priority orders $\prec^i$. Therefore, we define a composition operator on priority orders.

Definition 2.9 ($\oplus$) *Given two priority orders $\prec^1$ and $\prec^2$, we represent by $pr_1 \oplus pr_2$ the least priority order, if it exists, that contains $\prec^1 \cup \prec^2$. Otherwise, $\prec^1 \oplus \prec^2 = \bot$, where $\bot$ is an absorbing element with respect to $\oplus$: $(\prec \oplus \bot) = (\bot \oplus \prec) = (\bot \oplus \bot) = \bot$ for any priority order $\prec$.*

Two priority orders $\prec^1$ and $\prec^2$ for which $\prec^1 \oplus \prec^2 \neq \bot$ are called *compatible*. Notice that the operator $\oplus$ is associative and commutative.

Example 2.6 $\{a \prec_\infty b,\ c \prec_0 b\} \oplus \{c \prec_0 a\} = \{a \prec_\infty b,\ c \prec_0 a,\ c \prec_\infty b\}$*, that is, the sum contains its transitive hull.* $\{a \prec_\infty b,\ c \prec_0 b\} \oplus \{b \prec_0 c\} = \bot$*, since the transitive hull of the union of both priority orders is not irreflexive (it contains $b \prec_0 b$ and $c \prec_0 c$).*

We extend the operator $\oplus$ on priority functions.

Definition 2.10 ($\oplus$) *For two priority functions pr_1 and pr_2, we represent by $pr_1 \oplus pr_2$ the priority function defined as $\forall (s, \mathbf{x}) \in S \times \mathbb{R}^m_{\geqslant 0}\ .\ (pr_1 \oplus pr_2)(s, \mathbf{x}) = pr_1(s, \mathbf{x}) \oplus pr_2(s, \mathbf{x})$.*

The sum $pr_1 \oplus pr_2$ can be computed according to the following property, as can be easily verified.

Property 2.3 *Let $pr_1 = \{(C^i, \prec^i)\}_{i \in I}$ and $pr_2 = \{(C^j,\ \prec^j)\}_{j \in J}$ be two priority functions. Then,*

$$\begin{aligned}
pr_1 \oplus pr_2 \quad = \quad & \{(C^i \wedge C^j, \prec^i \oplus \prec^j) \mid i \in I \ \wedge\ j \in J \ \wedge\ (C^i \wedge C^j \neq \mathit{false})\} \ \cup \\
& \{(C^i \wedge \neg C^J, \prec^i) \mid i \in I \wedge (C^i \wedge \neg C^J \neq \mathit{false})\} \ \cup \\
& \{(C^j \wedge \neg C^I, \prec^j) \mid j \in J \wedge (C^j \wedge \neg C^I \neq \mathit{false})\}\ ,
\end{aligned}$$

where $C^I = \bigvee_{i \in I} C^i$, and $C^J = \bigvee_{j \in J} C^i$.

The timed system with priorities $\big((TS, pr), pr'\big)$ is defined as $(TS, pr \oplus pr')$. The fact that the priority function obtained as the sum of priority functions may be undefined for certain state constraints, plays a substantial role for verifying consistency of properties expressed by the composed priority functions, and will be extensively used in the following chapters.

Example 2.7 *Suppose that we want to restrict the timed system TS of example 2.5 according to two priority functions: pr as before, and* $pr_2 = \{(s \wedge (y < z), b \prec_0 c)\}$*. We have* $pr \oplus pr_2 = \{(s \wedge (y < z), \{a \prec_\infty b, b \prec_0 c, a \prec_\infty c\}),\ (s \wedge (y \geqslant z), \{a \prec_\infty b, c \prec_0 b\})\}$*, and compute the guards of TS′ as*

$$\begin{aligned}
g'_a &= g_a \wedge (y \geqslant z \vee (y > 5 \wedge z > 6)) \wedge (y < z \vee y > 5) \\
&= (x = 4 \wedge y > 5 \wedge (y \geqslant z \vee z > 6)) \\
g'_b &= g_b \wedge (y \geqslant z \vee z > 6) \ = \ (y = 5 \wedge (y \geqslant z \vee z > 6)) \\
g'_c &= g_c \wedge (y < z \vee y \neq 5) \ = \ (z \leqslant 6 \wedge (y < z \vee y \neq 5)) \ .
\end{aligned}$$

For $pr_3 = \{(s \wedge (y \leqslant z), b \prec_0 c)\}$*, that is, the priority order of* pr_2 *also applies for* $y = z$*, we have* $pr \oplus pr_3 = \{(s \wedge (y < z), \{a \prec_\infty b, b \prec_0 c, a \prec_\infty c\}),\ (s \wedge (y > z), \{a \prec_\infty b, c \prec_0 b\}),\ (s \wedge (y = z),\ \bot)\}$*, since the priority orders applying to* $s \wedge y = z$ *are not compatible. Therefore, s is considered as an error state in* $(TS, pr \oplus pr_3)$*, and has no leaving transitions according to definition 2.8.*

Before introducing the notion of *activity preservation* under priority choice, we need to characterize particular classes of X-constraints.

Definition 2.11 (Backward closure) *An* X*-constraint* C *is* backward closed *with respect to a rate vector* $\mathbf{b}$ *if* $\Diamond^{\mathbf{b}} C \implies C$.

Intuitively, when a backward-closed X-constraint C is *true*, then it has always been true. For instance, $x \leqslant 4$ is backward-closed for $\mathbf{b}[x] \in \{0, 1\}$, whereas $x \geqslant 4$ is not backward-closed for $\mathbf{b}[x] = 1$.

Definition 2.12 (Time convexity) *An* X*-constraint* C *is* time-convex *with respect to a rate vector* $\mathbf{b}$ *if*

$$\forall \mathbf{x} \in \mathbb{D}^m\ \forall d \in \mathbb{D}\ \big(C(\mathbf{x}) \wedge C(\mathbf{x} + d\mathbf{b}) \implies \forall d' \in \mathbb{D},\ d' < d\ .\ C(\mathbf{x} + d'\mathbf{b})\big)\ .$$

Time convexity corresponds to the usual notion of convexity, in the direction indicated by the rate vector $\mathbf{b}$. Any backward closed X-constraint is time-convex, whereas the converse is not true.

We understand by *deadlock* a system state from which no discrete transition will ever be enabled. Activity preservation guarantees that no deadlock is introduced by priority choice, that is, whenever the non prioritized timed system can eventually execute an action, then so can the timed system with priorities, provided that the priority function is defined for the current control state. Activity preservation has been shown in [Bor98, BS00] for priority functions $\{(true, \prec)\}$ where one priority order $\prec$ applies to all system states. The following proposition extends this result to priority functions where all $C^j(s)$ are backward closed for $s \in S$.

Proposition 2.1 (Activity preservation) *Let* $TS = (S, A, T, X, \mathbf{b}, h)$ *be a timed system,* $pr = \{(C^j, \prec^j)\}_{j \in J}$ *be a priority function where all* C^j *are backward closed, and* $(TS, pr) = (S, A, T', X, \mathbf{b}, h')$*. If for all* $t_i \in T$*,* $h(t_i) = (g_i, \tau_i, r_i)$*, and* $h'(t_i) = (g'_i, \tau_i, r_i)$*, then for all* $s \in S$ *such that* $\forall x \in \mathbb{D}^m \,.\, pr(s, \mathbf{x}) \neq \bot$*,*

$$\Diamond^{\mathbf{b}_s} \bigvee_{t_i \in T(s)} g'_i \quad = \quad \Diamond^{\mathbf{b}_s} \bigvee_{t_i \in T(s)} g_i \ .$$

Proof. See appendix A.

Example 2.8 *Coming back to example 2.7, activity preservation guarantees that if* s *can eventually be left in* TS*, then* s *can eventually be left in* $(TS, pr \oplus pr_2)$*, that is,* $\Diamond(g_a \vee g_b \vee g_c) = \Diamond(g'_a \vee g'_b \vee g'_c)$*. In fact,* $\Diamond(g_a \vee g_b \vee g_2) = \Diamond(x = 4 \vee y = 5 \vee z \leqslant 6) = (x \leqslant 4 \vee y \leqslant 5 \vee z \leqslant 6)$*, and*

$$\begin{aligned}
\Diamond(g'_a \vee g'_b \vee g'_c) &= \Diamond\Big(\big(x = 4 \wedge y > 5 \wedge (y \geqslant z \vee z > 6)\big) \ \vee \\
&\quad \big(y = 5 \wedge (y \geqslant z \vee z > 6)\big) \ \vee \ \big(z \leqslant 6 \wedge (y < z \vee y \neq 5)\big)\Big) \\
&= \Diamond\big(y = 5 \vee z \leqslant 6 \vee (x = 4 \wedge y > 5 \wedge z > 6)\big) \\
&= (x \leqslant 4 \ \vee \ y \leqslant 5 \ \vee \ z \leqslant 6) \\
&= \Diamond(g_a \vee g_b \vee g_c) \ .
\end{aligned}$$

The following, quite intuitive property says that a safety property ensured by a priority function pr is preserved when pr is composed with another priority function. It is a consequence of the *composability* property discussed in [AGS00], and is often used when modelling safety properties with priority functions.

Property 2.4 (Safety preservation) *Let* $TS = (S, A, T, X, \mathbf{b}, h)$ *be a timed system,* pr *and* pr' *be priority functions, and* $t = (s, a, s') \in T$ *be a transition of* $(TS, pr \oplus pr')$ *with* $h(t) = (g, \tau, r)$*. Moreover, let* g' *denote the guard of* t *in* (TS, pr)*, and* g'' *the guard of* t *in* $(TS, pr \oplus pr')$*. Then we have for any clock valuation* $\mathbf{x} \in \mathbb{D}^m$*,* $g''(\mathbf{x}) \implies g'(\mathbf{x})$*, that is, if* t *is disabled in* (TS, pr)*, then it remains disabled in* $(TS, pr \oplus pr')$*.*

Proof. If $g'(\mathbf{x}) = \mathit{false}$, then we have by definition 2.8 and the fact that $pr(s) \subseteq pr(s) \oplus pr'(s)$, that $g''(\mathbf{x}) = \mathit{false}$. If $g'(\mathbf{x}) = \mathit{true}$, then the above implication is trivially verified. ■

2.3 Parallel Composition

We define a parallel composition operator $\|_\Sigma$ similar to that studied in [BS00], which we extend to timed systems.

Definition 2.13 (Parallel composition) *Let* (TS_i, pr_i) *be a timed system with priorities with* $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$ *for* $i \in \{1, \ldots, n\}$*. We suppose that the timed systems have disjoint action sets. Moreover, let* Σ *be a set of subsets of* $\bigcup_{i=1}^n A_i$ *such* *that any* $\mathbf{a} \in \Sigma$ *contains at most one action of each component, and* $|\mathbf{a}| \geqslant 2$. *The* parallel composition *of the timed systems with priorities* (TS_i, pr_i), $i \in \{1, \dots, n\}$, *is the timed system with priorities* (TS, pr) *with* $TS = (S, A, T, X, \mathbf{b}, h)$ *defined by*

- $S = S_1 \times \dots \times S_n$; $\quad A = \bigcup_{i=1}^{n} A_i \,\cup\, \Sigma$; $\quad X = \bigcup_{i=1}^{n} X_i$. *Non-synchronizing actions* $a \in A \setminus \Sigma$ *keep their controllability, whereas synchronization actions* $\mathbf{a} \in \Sigma$ *are controllable if and only if all elements of* $\mathbf{a}$ *are controllable;*
- $\forall s = (s_1, \dots, s_n), s' = (s'_1, \dots, s'_n) \in S$.
  - *interleaving transitions:* $\forall i \in \{1, \dots, n\}\ \forall a \in A_i \,.\, t = (s, a, s') \in T \iff$
    $t_i = (s_i, a, s'_i) \in T_i \,\wedge\, \forall j \in \{1, \dots, n\} \setminus \{i\} \,.\, s'_j = s_j$;
    $h(t) = h_i(t_i)$;
  - *synchronization transitions:* $\forall \mathbf{a} \in \Sigma \,.\, t = (s, \mathbf{a}, s') \in T \iff$
    $\forall i \in \{1, \dots, n\} \,.\, \big(t_i = (s_i, a_i, s'_i) \in T_i \,\wedge\, a_i \in \mathbf{a} \,\vee\, s_i = s'_i \,\wedge \mathbf{a} \cap A_i = \emptyset\big)$;
    if for any synchronizing transition t_i, $h_i(t_i) = (g_i, \tau_i, r_i)$,
    then $h(t) = (\bigwedge_{a_i \in \mathbf{a}} g_i,\ \max_{a_i \in \mathbf{a}}\{\tau_i\},\ \bigcup_{a_i \in \mathbf{a}} r_i)$, *where* $\lambda < \delta < \epsilon$;
- *for* $s = (s_1, \dots, s_n) \in S$ *and* $x \in X$, $\mathbf{b}_s[x] = \min_{x \in X_i} \mathbf{b}_{i,s_i}[x]$;

and $pr = \bigoplus_{i=1}^{n} pr_i \oplus pr_\Sigma$ *with*

$$\begin{aligned} pr_\Sigma \;=\; & \big\{\big(s, \{a \prec_\infty \mathbf{a}\}\big) \mid (s, \mathbf{a}, s') \in T \,\wedge\, \mathbf{a} \in \Sigma \,\wedge\, a \in \mathbf{a}\big\} \,\oplus \\ & \big\{\big(s, \{\mathbf{a_1} \prec_\infty \mathbf{a_2}\}\big) \mid (s, \mathbf{a_1}, s_1), (s, \mathbf{a_2}, s_2) \in T \,\wedge\, \mathbf{a_1}, \mathbf{a_2} \in \Sigma \,\wedge\, \mathbf{a_1} \subset \mathbf{a_2}\big\} \,. \end{aligned}$$

According to this definition, a *synchronization action* $\mathbf{a}$ is enabled when all *synchronizing actions* $a_i \in \mathbf{a}$ are enabled. All timed systems that do not synchronize in $\mathbf{a}$ remain idle. Action $\mathbf{a}$ is as urgent as the most urgent of the synchronizing actions, that is, one of the synchronizing actions of $\mathbf{a}$ becoming urgent while $\mathbf{a}$ is enabled makes $\mathbf{a}$ become urgent, too.

In addition to the priority functions pr_i of the composed timed systems, the priority function pr_Σ is used to achieve *maximal progress* [HR95], ensuring that interleaving transitions can be taken only when synchronization is not possible, and that among the possible synchronization transitions, the maximal one is taken.

Hypothesis: In order to guarantee that uncontrollable actions are never disabled by maximal progress, we assume that an uncontrollable action a^u may only synchronize with actions labelling transitions that are guarded by *true*, such that the synchronization is enabled whenever a^u is disabled by maximal progress.

We write $(TS, pr) = \|_\Sigma \big\{(TS_i, pr_i) \mid i \in \{1, \dots, n\}\big\}$, and $a_i \shortmid \dots \shortmid a_j$ for the synchronization action $\{a_i, \dots, a_j\} \in \Sigma$.

Remark 2.1 *It has been shown in [BS00] that the parallel composition operator* $\|$ *is associative and commutative if we assume that* $\{a_1, \dots, a_k\} \shortmid \{b_1, \dots, b_l\} = \{a_1, \dots, a_k, b_1, \dots, b_l\}$. *Therefore, we have*

$$\begin{aligned} & \big\|_\Sigma \big\{(TS_1, pr_1),\ \|_{\Sigma_{23}}\{(TS_2, pr_2), (TS_3, pr_3)\}\big\} \\ & \quad = \ \|_{\Sigma \cup \Sigma_{23}}\{(TS_1, pr_1), (TS_2, pr_2), (TS_3, pr_3)\} \,, \end{aligned}$$

that is, hierarchical parallel composition is equivalent to parallel composition of all components at once.

When reasoning about timed systems obtained by parallel composition, we extend a given predicate $\mathcal{P}$ on control states of some component system TS_i to a predicate on control states of the product system by interpreting $\mathcal{P}$ on S as $\mathcal{P}(s_1, \ldots, s_n) \iff \mathcal{P}(s_i)$.

Example 2.9 *Consider the timed system* `buffer` *of figure 2.5. Transition* refresh

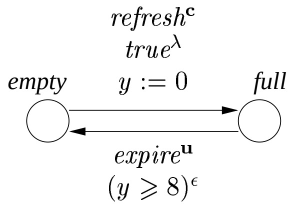


Figure 2.5: Timed system `buffer` modelling a buffer with freshness constraint.

from state empty *to state* full *resetting clock y models the writing of some data to the buffer. After a delay of 8 time units, transition* expire *becomes urgent and makes* `buffer` *return to state* empty*, modelling the freshness constraint that data that are older than 8 time units are discarded.*

Let us compose the cyclic process `P` *of example 2.2 with* `buffer` *so that* `P` *serves as a producer that makes* `buffer` *pass from* empty *to* full *when* `P` *takes the transition* write*; if* `buffer` *is already* full *at that moment, then the transition* write *must be possible without changing the state of* `buffer` *— the produced item is discarded.*

This cooperation between `P` *and* `buffer` *is achieved by synchronizing the actions* write *and* refresh*. Formally, $TS = \|_\Sigma\{\mathtt{P}, \mathtt{buffer}\}$ with $\Sigma = \{write \mid refresh\}$. The priority function pr_Σ guaranteeing maximal progress is $pr_\Sigma = \{\big((computing,\ empty),\ \{write \prec_\infty write \mid refresh, refresh \prec_\infty write \mid refresh\big)\}$. Figure 2.6 shows TS. Notice the transitions leaving state (computing, empty): the interleaving transitions* write *and* refresh *are disabled if the synchronization transition* write|refresh *is possible.*

However, the obtained product system is not yet satisfying, since the interleaving transitions refresh *from the states (sleeping, empty) and (waiting, empty) are still enabled. This problem can be resolved by applying the priority function*

$$\begin{aligned} pr \;=\; & \{\big((sleeping,\ empty),\ refresh \prec_\infty arrive\big), \\ & \big((waiting,\ empty),\ refresh \prec_\infty compute\big)\} \end{aligned}$$

to $\|_\Sigma\{\mathtt{P}, \mathtt{buffer}\}$ which disables the interleaving refresh *transitions. It is easy to verify that pr is compatible with the priority function pr_Σ, since both functions apply to disjoint sets of states. Hence, activity preservation is guaranteed, and we obtain as the sum of both priority functions $pr_\Sigma \oplus pr = pr_\Sigma \cup pr$.*

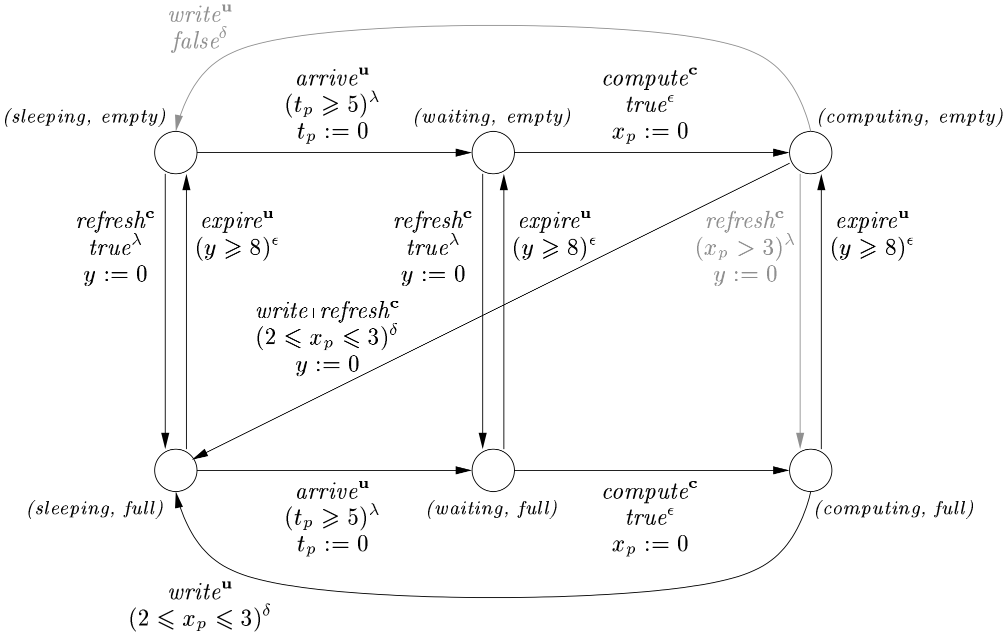


Figure 2.6: The product *TS* of `P` and `buffer` with $\Sigma = \{write \shortmid refresh\}$.

The synchronization paradigm of definition 2.13 satisfies maximal progress as well as the requirement of *activity preservation*, that is, if in one of the components an action is eventually enabled, then it is eventually enabled in the product, too. This result has been shown in [BS00]. Intuitively, adding a synchronization transition does not introduce any deadlock, and proposition 2.1 guarantees activity preservation under the priority function pr_Σ ensuring maximal progress.

In the untimed case, both of the paradigms CCS *(Calculus of Communicating Systems)* [Mil89] and CSP *(Communicating Sequential Processes)* [Hoa85] satisfy one requirement each: CSP partitions actions into interleaving actions and synchronizing actions, which ensures that synchronizing actions do not interleave, at the risk of introducing deadlocks. In CCS, all actions interleave, which ensures activity preservation. A global restriction operator allows to cut off interleaving transitions, again at the risk of introducing deadlocks.

Parallel composition with priorities, still in the untimed case, is dealt with in [BBK86, CH90, CW95, CLN98, Phi01], and in the timed framework of the process algebra ACSR [BGL97], and timed extensions of CCS [BCL99, CLN01].

Action constraints

In order to prune off certain transitions of a timed system, we use a particular case of *action constraints* as introduced in [AGS00]. Let $TS = (S, A, T, X, \mathbf{b}, h)$ be a timed system, and $K \subseteq 2^S \times 2^A$ be an action constraint associating with a set of control states, actions that are to be cut off. The timed system restricted by K, $TS' = TS/K$, is defined as $TS' = (S, A, T', X, \mathbf{b}, h')$, where $T' = \{(s, a, s') \in T \mid \forall (S', A') \in K \,.\, s \notin S' \vee a \notin A'\}$. h' is the restriction of h to T'. The restriction of a timed system with priorities by an action constraint is defined as $(TS, pr)/K = (TS/K, pr)$.

Example 2.10 (Blocking rendez-vous) *According to definition 2.13, synchronizing actions can interleave in control states where the synchronization action will never become enabled. Synchronizing actions that are not in* Σ *are prevented from interleaving by restricting the product timed system* $TS = (S, A, T, X, \mathbf{b}, h)$ *by the action constraint* $K = S \times \{a \in A \setminus \Sigma \mid \exists \mathbf{a} \in \Sigma \,.\, a \in \mathbf{a}\}$.

2.4 Related Models

The formalism of timed systems with priorities defined above is the framework that will be used throughout this report. However, alternative formalisms have been studied that are adapted for particular modelling or verification purposes, and some of them have been implemented in the tool presented in chapter 6. In this section, we briefly discuss some of these variants.

2.4.1 Timed Systems with Deadlines

Timed systems with deadlines are a variant of timed systems as in definition 2.2, where the labelling function h associates with each untimed transition an X-constraint called *deadline*, instead of an urgency type, such that if $h(t_i) = (g_i, d_i, r_i)$ for some $t_i \in T$, then $d_i \implies g_i$. As to the semantics of a timed system with deadlines, the time progress condition of definition 2.3 becomes

$$(s, \mathbf{x}) \xrightarrow{d} (s, \mathbf{x} + d\mathbf{b}_s) \iff \bigwedge_{t_i \in T(s)} \forall d' \in \mathbb{D},\ d' < d \,.\, \neg d_i(\mathbf{x} + d'\mathbf{b}_s)\ .$$

Timed systems are at least as expressive as timed systems with deadlines: any timed system with deadlines can be transformed into a timed system by splitting each transition $t = (s, a, s')$ and $h(t) = (g, d, r)$ into a lazy transition $t^\lambda = (s, a^\lambda, s')$ with $h(t^\lambda) = (g \wedge \neg d, \lambda, r)$, and an eager transition $t^\epsilon = (s, a^\epsilon, s')$ with $h(t^\epsilon) = (d, \epsilon, r)$.

As for timed systems, a priority function can be associated with a timed system with deadlines: for $t = (s, a, s') \in T$ with $h(t) = (g, d, r)$, $h'(t) = (g', d \wedge g', r)$, where g' is obtained by restricting g as in definition 2.8. However, the semantics of urgency is somewhat counter-intuitive in this model. Consider for example two transitions $t_1 = (s, a_1, s_1)$ with $h(t_1) = (x \leqslant 3, x = 3, \emptyset)$, and $t_2 = (s, a_2, s_2)$ with $h(t_2) = (y \geqslant 5, \mathit{false}, \emptyset)$. a_1 has its deadline at the falling edge of its guard, which means that the action must be taken before it becomes disabled. If we apply the priority function $pr = \{(\mathit{true}, \{a_1 \prec_1 a_2\})\}$ to this timed system, the guard of a_1 is restricted to $x \leqslant 3 \wedge \neg\Diamond_1 y \geqslant 5$, that is, $x \leqslant 3 \wedge y < 4$, and the deadline is accordingly restricted to $x = 3 \wedge y < 4$. From state $(s, x = 2 \wedge y = 3)$ where a_1 is enabled, time can pass until state $(s, x = 4 \wedge y = 5)$ is reached where a_1 is disabled — action a_1 does not correspond to a delayable action any more.

2.4.2 Timed Automata

It is sometimes useful to translate a timed system into a timed automaton in order to use tools that are based on the formalism of timed automata, such as UPPAAL and KRONOS.

A *timed automaton* is a tuple $TA = (S, A, T, X, I, h)$, where S, A, T, and X are as in definition 2.2; I is a function that associates with each control state $s \in S$ an X-constraint $I(s)$ called *invariant*, and the function h associates with each untimed transition $t \in T$ a pair $h(t) = (g, r)$ composed of a guard and a set of clocks to be reset. Both guards and invariants are generated by the grammar

$$\mathcal{P} ::= \mathit{true} \mid x \lhd c \mid x - y \lhd c \mid \mathcal{P} \wedge \mathcal{P} \mid \neg\mathcal{P} \ ,$$

where $x, y \in X$, $\lhd \in \{\leqslant, <\}$, and $c \in \mathbb{Z}$. For any state $(s, \mathbf{x}) \in S \times \mathbb{D}^m$,

- $(s, \mathbf{x}) \xrightarrow{a} (s', \mathbf{x}[r])$ if $(s, a, s') \in T$ with $h(s, a, s') = (g, r)$, $g(\mathbf{x}) = \mathit{true}$, and $I(s)(\mathbf{x}) = I(s')(\mathbf{x}[r]) = \mathit{true}$;
- for any delay $d \in \mathbb{D}$, $(s, \mathbf{x}) \xrightarrow{d} (s, \mathbf{x} + d\mathbf{1}) \iff \forall d' \in \mathbb{D},\ d' \leqslant d \ .\ I(s)(\mathbf{x} + d'\mathbf{1})$, where $\mathbf{1}$ is the m-dimensional vector $(1, \ldots, 1)$.

Translating a timed system into a timed automaton

Any timed system $TS = (S, A, T, X, \mathbf{b}, h)$ where $\mathbf{b}_s[x] = 1$ for all $s \in S$ and all $x \in X$, and all eager guards are time-convex, can be transformed into a timed automaton $TA = (S', A, T', X \cup \{z\}, I, h')$ with the same behaviour as TS. $z \notin X$ is a new clock. TA is obtained by splitting the control states of TS according to the urgency of the outgoing transitions. The following definition is used to compute the invariants of the split control states.

Definition 2.14 (Right-closure) *For an X-constraint C and a rate vector $\mathbf{b}$, we define the* right-closure *of C with respect to $\mathbf{b}$ as $\forall \mathbf{x} \in \mathbb{R}^m_{\geqslant 0}$,*

$$\mathit{close\_right}_{\mathbf{b}}(C)(\mathbf{x}) \iff C(\mathbf{x}) \ \vee\ \forall d > 0\ \exists \mathbf{x}' \in \mathbb{D}^m\ \exists d' < d\ .\ \mathbf{x}' + d'\mathbf{b} = \mathbf{x}\ \wedge\ C(\mathbf{x}')$$

and omit $\mathbf{b}$ if $\mathbf{b}[x] = 1$ for all clocks $x \in X$.

Notice that for $\mathbb{D} = \mathbb{N}$, the right-closure of any X-constraint C is equal to C.

Each control state in $s \in S$ is split according to invariants depending on the guards and the urgency types of the transitions leaving s. For any $t \in T(s)$ with $h(t) = (g, \tau, r)$ and $g \neq \mathit{false}$, the invariants of the split control states in TA are:

$\tau = \delta$: $\Diamond g$, and $\neg\Diamond g$;

$\tau = \epsilon$: $\mathit{close\_right}(\neg g \wedge \Diamond g)$, $g \wedge (z = 0)$, and $\neg\Diamond g$. Notice that the invariant $\mathit{close\_right}(\neg g \wedge \Diamond g)$ contains the rising edge of g, which means that the state must be left as soon as g becomes true.

z is reset by the transitions entering the state with invariant $g \wedge (z = 0)$. Intersecting its invariant g with $z = 0$ models the transient (urgent) character of this state: it must be left without letting time pass.

$\tau = \lambda$: no splitting.

The entering transitions are restricted such that their post-condition implies the invariant of the split state they enter. More precisely, the guard of a transition resetting the clocks in r and entering a state with invariant I, is intersected with $I[r^{-1}]$, denoting the clock valuations $\mathbf{x} \in \mathbb{R}^m_{\geqslant 0}$ that satisfy I when the clocks in r are reset: $(I[r^{-1}])(\mathbf{x}) \iff I(\mathbf{x}[r])$. Leaving transitions whose guard is disjoint with the invariant, can be discarded in *TA*.

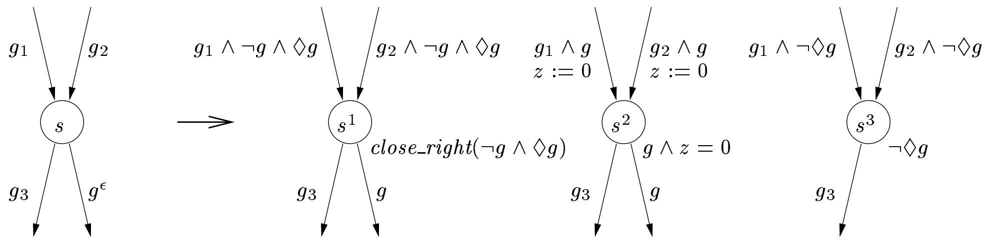


Figure 2.7: Representing an eager guard g^ϵ in terms of invariants.

Figure 2.7 illustrates the splitting of a state s according to an eager guard g^ϵ, under the simplifying assumption that the actions in A do not reset any clocks in *TS*.

It is easy to see that after each step of the splitting procedure, for a given transition t of *TS* entering s, the guard of t is equal to the disjunction of the guards obtained by splitting t. We sketch a proof showing that *TS* and *TA* are equivalent modulo the split states. By construction, for any transition t entering some control state s^i in *TA*, the invariant $I(s^i)$ holds for the postcondition of t. Consider some state $(s, \mathbf{x}) \in S \times \mathbb{D}^m$ of *TS*. There exists some $s^i \in S'$ such that $I(s^i)(\mathbf{x})$. By construction of the invariants, time can pass in *TA* if and only if none of the leaving transitions has become urgent in *TS*. *TA* can take a transition (s^i, a, s^j) if and only if *TS* can take a transition (s, a, s'), where s^j has been obtained by splitting s'.

Although the size of *TA* may grow exponentially with the number of transitions in T, practice shows that the size of *TA* often remains of the order of *TS*, due to transitions whose guards become *false* during splitting of the state they are entering, or that become unreachable from the state they are leaving. This is especially the case for *structurally live* timed systems, which are discussed in chapter 4.

2.4.3 Discrete-Time Interpretation

According to definition 2.3, a given timed system *TS* can be interpreted over $\mathbb{D} = \mathbb{R}_{\geqslant 0}$ in a continuous time model, as well as in a discrete-time model by taking $\mathbb{D} = \mathbb{N}$. The syntactical structure of *TS* remains the same; only the domains of system states and the instants when transitions become urgent, change. The same remark holds for timed systems with deadlines, and timed automata.

There are two main reasons why it can be interesting to interpret a given model in discrete rather than continuous time. First, for timed systems where clocks can be stopped, certain properties are in general undecidable in a continuous time interpretation [KPSY92], and become decidable in discrete time if clocks are bounded. The second reason is rather practical: a discrete-time interpretation allows to use verification and synthesis tools based on a discrete-time model, such as [FGK+96], which sometimes allows to tackle problems that are a magnitude larger [BMT99].

2.4.4 Synchronization modes

The definition 2.13 of parallel composition is kept voluntarily basic. However, extensions are possible and useful to allow more powerful synchronization primitives.

In definition 2.13, the guard of a synchronization action is defined as the intersection of the guards of the synchronizing actions. This is a widely used synchronization paradigm: the synchronization action can take place if all synchronizing actions are enabled.

In order to model particular application domains such as multimedia documents, it is convenient to provide different synchronization modes [BS97, Gößl98, BS00]. This can be done by associating with a parallel composition a function $\mathit{mode} : \Sigma \to \{\mathit{AND}, \mathit{OR}, \mathit{MIN}, \mathit{MAX}\}$ defining for each synchronization action $\mathbf{a} \in \Sigma$ its mode $\mathit{mode}(\mathbf{a})$. *AND* is the mode described above; *OR* means that the synchronization can take place as soon as one of its actions is enabled, which models the interruption of parallel processes by the first process that terminates. A *MIN* synchronization is enabled if one of the synchronizing actions is enabled, and the others will become enabled some time in the future, whereas a *MAX* synchronization is enabled if one of its actions is enabled, and all other actions have already been enabled in the past, which models waiting for the last process to arrive.

Let $t = (s, \mathbf{a}, s')$ be a synchronization of the transitions $t_i = (s_i, a_i, s'_i)$ with $a_i \in \mathbf{a}$ and $h(t_i) = (g_i, \tau_i, r_i)$. Then, $h(t) = (g, \tau, r)$, where the τ and r are as in definition 2.13, and the guard g of t is computed according to the synchronization mode:

$$g = \begin{cases} \bigwedge_{a_i \in \mathbf{a}} g_i & \text{if} \quad \mathit{mode}(\mathbf{a}) = \mathit{AND} \\ \bigvee_{a_i \in \mathbf{a}} g_i & \text{if} \quad \mathit{mode}(\mathbf{a}) = \mathit{OR} \\ \bigvee_{a_i \in \mathbf{a}} \left(g_i \wedge \bigwedge_{a_j \in \mathbf{a} \setminus \{a_i\}} \Diamond^{\mathbf{b}_s} g_j\right) & \text{if} \quad \mathit{mode}(\mathbf{a}) = \mathit{MIN} \\ \bigvee_{a_i \in \mathbf{a}} \left(g_i \wedge \bigwedge_{a_j \in \mathbf{a} \setminus \{a_i\}} \ominus\!\!\!\!\Diamond^{\mathbf{b}_s} g_j\right) & \text{if} \quad \mathit{mode}(\mathbf{a}) = \mathit{MAX}\,. \end{cases}$$

Example 2.11 *Suppose that process* P *of example 2.2 is to be composed, as a producer, directly with a consumer process, without an intermediate buffer, such that the first of the processes to be ready to read or write waits for the other process to be ready as well. This is done by synchronizing the action* write *of* P *with the corresponding action, say* read, *of the consumer, using MAX synchronization. The guard of* write *is* $g_w = (2 \leqslant x_p \leqslant 3 \wedge t_p \leqslant 5)$. *If we assume the guard of* read *to be* $g_r = (1 \leqslant x_c \leqslant 4)$, *then the guard of the synchronization action read|write is*

$$\begin{aligned} g_{rw} &= g_w \wedge \diamondsuit g_r \ \vee \ \diamondsuit g_w \wedge g_r \\ &= 2 \leqslant x_p \leqslant 3 \wedge t_p \leqslant 5 \wedge 1 \leqslant x_c \ \vee \ 2 \leqslant x_p \wedge -3 \leqslant x_p - t_p \leqslant 3 \wedge 1 \leqslant x_c \leqslant 4 \ . \end{aligned}$$

If the same synchronization semantics was to be expressed in terms of AND synchronization, the structure of both processes would have to be modified.

Using these synchronization modes can drastically increase the concision of the timed system in certain cases [BST98], and improve readability of the model.

2.4.5 Petri Nets with Urgency

The input language of the tool PROMETHEUS, which is presented in chapter 6, is based on Petri nets with urgency. Their semantics is closely related to that of timed systems.

Definition 2.15 (Petri net with urgency) *A* Petri net with urgency *is a tuple* $(P, P_0, A, T, X, \mathbf{b}, h)$ *where*

- P *is a finite set of* places*;*
- $P_0 \subseteq P$ *is the* initial marking*;*
- A *is a finite vocabulary of* actions*, partitioned into two sets of controllable actions* A^c *and uncontrollable actions* A^u*;*
- $T \subseteq 2^P \times A \times 2^P$ *is a set of* transitions*;*
- $X = \{x_1, \dots, x_m\}$ *is a set of clocks;*
- $\mathbf{b} : P \to \{0,1\}^m$ *is a function mapping each place on a boolean* rate vector*; and*
- h *is a function labelling each transition* $t \in T$ *with a tuple* $h(t) = (g, \tau, r)$*, where* g *is a* guard*,* $\tau \in \{\lambda, \delta, \epsilon\}$ *is an* urgency type*, respectively* lazy*,* delayable*, and* eager*, and* $r \subseteq X$ *is a set of clocks to be* reset*.*

We define the semantics of a 1-safe Petri net with urgency in terms of the timed system representing its marking graph.

Definition 2.16 (Semantics of a Petri net with urgency) *Let* $N = (P, P_0, A, T, X, \mathbf{b}, h)$ *be a 1-safe Petri net with urgency. The* marking graph *of* N *is the timed system* $TS = (S, A, T', X, \mathbf{b}', h')$ *defined as follows:*

- $S \subseteq 2^P$ *is the set of markings reachable from* P_0 *in the untimed Petri Net* (P, P_0, A, T)*;*
- $T' = \big\{t' = (s, a, s') \mid \exists t = (P_1, a, P_2) \in T \ . \ P_1 \subseteq s \ \wedge \ (s \setminus P_1) \cap P_2 = \emptyset \ \wedge \ s' = (s \setminus P_1) \cup P_2\big\}$*, and* $h'(t') = h(t)$*;*
- *for any* $s \in S$ *and* $x \in X$*,* $\mathbf{b}'(s)[x] = \min_{p \in s} \mathbf{b}(p)[x]$.

Let $\rightarrow^{TS}$ denote the transition relation of TS, as in definition 2.3. The transition relation $\rightarrow^{N}$ of N is defined as follows: if N is in state $(s, \mathbf{x}) \subseteq 2^P \times \mathbb{D}^m$, then for any $\tau \in A \cup \mathbb{D}$, $(s, \mathbf{x}) \xrightarrow{a}{}^{N} (s', \mathbf{x}') \iff (s, \mathbf{x}) \xrightarrow{a}{}^{TS} (s', \mathbf{x}')$.

The function h' is well-defined: if for some $t' = (s, a, s') \in T'$ of the timed system there are two transitions $t_1 = (P_1, a, P_1')$ and $t_2 = (P_2, a, P_2')$ of the Petri net such that $P_i \subseteq s$, $(s \setminus P_i) \cap P_i' = \emptyset$, and $s' = (s \setminus P_i) \cup P_i'$ for $i \in \{1, 2\}$, then it follows that $s \setminus P_1 = s \setminus P_2$, $P_1 = P_2$, and $P_1' = P_2'$, and therefore $t_1 = t_2$.

2.5 About Logical and Physical Time

When modelling real-time systems, the choice of how time and time progress are modelled, is crucial for the validity of the model: if the model of time ("logical time") does not reflect certain properties of physical time that are important for the correctness of the model, or introduces artifacts that do not exist in physical time, then analysis of the model may yield results that do not correspond to the behaviour of the modelled system. Therefore — similarly to any other modelling task — one has to decide on what are the desired properties of logical time, find an appropriate model, and study the possible artifacts introduced by this model. In the case of modelling real-time systems, some desired properties of logical time are:

- logical time (measured by an imaginary reference clock that is never reset nor stopped) cannot be stopped (*timelock-freedom*);
- logical time diverges beyond any bound (*non-zenoness*);
- within any finite amount of logical time, only a finite number of actions can occur (*strong non-zenoness*), according to the observation that any physical process, no matter how fast, cannot be infinitely fast.

Violations of these requirements by a model can be considered as modelling errors.

In the real world — at least if reasoning in classical physics —, events occur as an effect caused by precedent events, according to the natural laws; consider for example the impact of a stone on the ground, some time after dropping it from a certain height. Within the model of a real-time system, natural laws are generally not modelled explicitly. Therefore, the model of time must provide some means to let *time* enable, enforce, or disable the occurrence of actions.

In the formalism of timed automata, this is what invariants are for: the system can remain in the current control state as long as the associated invariant is *true*, but must take an action before the invariant becomes *false*. Therefore, care has to be taken to avoid timelock-prone specifications where the invariant of a state can become *false* while no transition leaving that state is enabled. This is especially the case when transitions are to be disabled, for example in a parallel composition of timed automata.

In the formalism of timed systems, urgency is directly encoded in the type of transitions; as soon as any transition becomes urgent, the triggering of an action has priority over time progress. Associating urgency with transitions nearly rules out the risk of timelocks, since logical time cannot be blocked unless an action is enabled. Yet, there is one source of timelocks: when an eager guard is not left-closed (say, $5 < x$), then time may progress up to, but not beyond 5, which means that the guard will never become enabled. A class of timed systems ruling out this remaining risk of timelocks and preserving the above requirements under priority choice and parallel composition, is discussed in chapter 4.

Chapter 3

Modelling Real-Time Applications

In this chapter, a methodology for modelling a real-time applications as a timed system with priorities, is provided. The real-time application model is built, following an incremental description principle, from the timed systems representing the processes, and a modular description of their interaction.

The methodology provides guidelines for building abstractions of real-time applications relevant to scheduling issues, which are the subject of chapter 5. We focus on reactive systems, that is, the processes are cyclic and do not terminate. Processes are specified by their untimed transition system including information about the use of resources, and timing constraints relating in particular process execution speed with the dynamics of their external environment.

Interaction between the processes, in particular the functional aspects of resource management, is modelled by parallel composition, and a set of priority functions. They describe in a modularized manner mutual exclusion on shared resources, preemption, synchronous communication between the processes, and atomicity of action sequences. The model of interaction is essentially obtained as the composition of the priority functions modelling the behavioural aspects mentioned above. It can be considered as a synchronization layer above the model of the processes, as shown in figure 1.1.

3.1 Modelling Processes

We consider processes (sometimes also called tasks in the literature) whose relative deadline is inferior or equal to the inter-arrival time, so that we do not have to distinguish several instances (or jobs) of the same process. Therefore, each process can be modelled as a timed system with a finite set of control states. Hereafter, we show how to model cyclic processes that can exist in a bounded number of instances, for example if their relative deadline is greater than the inter-arrival time.

Resources

The processes can share common *resources*, which can be physical (such as a processor, memory, or a communication bus) or logical (such as data structures). Each resource is either *preemptable* or *non-preemptable*. A process using a preemptable resource can be *preempted* from that resource in order for the resource to be granted to another process and to be given back to the preempted process later. If a non-preemptable resource is granted to a process, than the latter keeps the resource until completion of its using. We denote by R the set of resources, and partition R into the set of preemptable resources R_p, and the set of non-preemptable resources R_n.

We distinguish between the process to be modelled, and the timed system modelling it. Let P_i be a process to be modelled by the timed system $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$. We first give a classification of the discrete transition system (S_i, A_i, T_i), and then add the timing constraints according to the specification of process P_i.

Control states

For a given resource $r \in R$, we define two predicates on the control states of TS_i specifying the use of r by P_i:

- $U_{i,r}(s)$ meaning that process P_i is *using* r in state $s \in S_i$, and
- $W_{i,r}(s) \iff \neg U_{i,r}(s) \wedge \exists (s, a, s') \in T_i \;.\; U_{i,r}(s')$ meaning that process P_i is *waiting* for r in state $s \in S_i$. That is, s is a waiting state if there is a transition issued from s leading to a using state.

States s that are neither using nor waiting states are called *sleeping* states, characterized by $\neg \bigvee_{r \in R} \big(U_{i,r}(s) \vee W_{i,r}(s)\big)$. In order to cope with scheduling issues, it is sufficient to distinguish the above state classes and to make abstraction from individual states that do not carry additional information pertinent to resource management. We will discuss the modelling of scheduling policies in chapter 5.

By slight abuse of the notations, we us the predicates $U_{i,r}$ and $W_{i,r}$ as sets of states and write for example $s \in U_{i,r}$ if $U_{i,r}(s)$.

Example 3.1 *Suppose that the process* `P` *of example 2.2 uses the resource CPU in state* computing, *that is,* $U_{\mathtt{P},CPU} = \{computing\}$*. It follows that* $W_{\mathtt{P},CPU} = \{waiting\}$*, and* sleeping *is the only sleeping state of* `P`.

Example 3.2 *If process* `P` *of example 2.2 can alternatively be executed on any of the processors* CPU_1 *and* CPU_2*,* `P` *is modelled by the timed system shown in figure 3.1, with* $U_{\mathtt{P},CPU_1} = \{computing_1\}$*, and* $U_{\mathtt{P},CPU_2} = \{computing_2\}$*. Timing information has been omitted in the figure for the sake of simplicity.*

Actions

We distinguish the following types of actions for process P_i:

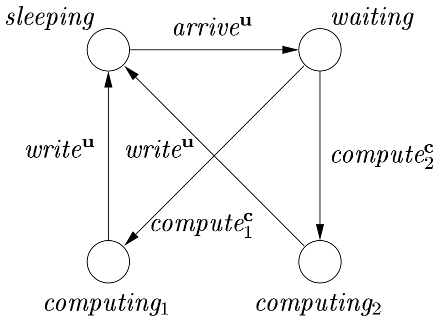


Figure 3.1: Process P with computation on one of two processors.

arrival actions leading from a sleeping to a waiting state;

requesting actions for some resource r, leading from a state $s \notin W_{i,r}$ to a state $s' \in W_{i,r}$;

begin actions for some resource r leading from a state $s \in W_{i,r}$ to a state $s' \in U_{i,r}$. $BGN_{i,r}$ denotes this set of actions. Begin actions are usually eager to denote the fact that a process takes the resource as soon as it is allowed to do so;

end actions for some resource r, leading from a state $s \in U_{i,r}$ to a state $s' \notin U_{i,r}$. $END_{i,r}$ denotes this set of actions;

timeout actions leading from a state $s \in W_{i,r}$ to a state $s' \notin W_{i,r} \cup U_{i,r}$.

Notice that arrival and begin actions are disjoint, as well as arrival and end actions. Furthermore we assume that begin and end actions are disjoint. Figure 3.2 recapitulates the different types of actions.

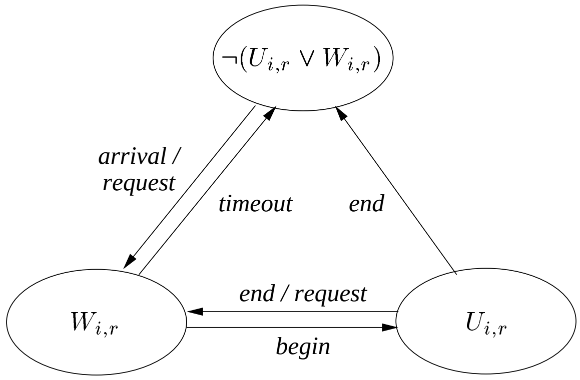


Figure 3.2: Action types with respect to a given resource $r \in R$.

The decision when to grant a resource to a process is up to the scheduler, whereas the decision when to free a resource belongs to the process. Therefore, begin actions are considered as controllable, whereas end and arrival actions are uncontrollable.

Example 3.3 *In example 2.2, the action* arrive *of process* `P` *is indeed an arrival action, and thus uncontrollable, since it leads from a sleeping to a waiting state. The action* compute *leading from a waiting to a using state is a begin action and thus controllable, and the action* write*, leading from a using to a sleeping state, is an end action and uncontrollable.*

Hypothesis: We suppose that any begin action acquiring a preemptable resource $r \in R_p$ does not acquires any other preemptable resource $r' \in R_p$. Atomic allocation of more than one preemptable resource can be modelled using the atomicity operator introduced below.

Timing constraints

We associate with each process P_i the following clocks:

- A clock t_i measuring the time elapsed since the arrival of P_i. Therefore, t_i is reset by arrival actions and has its rate everywhere equal to one.
- For each resource $r \in R$ used by P_i, a clock $x_{i,r}$ is introduced to measure execution time. This clock is reset at all begin actions acquiring r, and has its rate equal to 1 at states s such that $U_{i,r}(s)$.

The clocks are used to express the following classes of timing constraints.

Inter-arrival constraints. These constraints conjunct guards of arrival actions. Usually, they are conditions of the form $T_i^l \leqslant t_i \leqslant T_i^u$, where $T_i^l > 0$ and T_i^u are lower and upper bounds of inter-arrival times. For P_i strictly periodic, take $T_i^l = T_i^u$, and for P_i sporadic take $T_i^u = \infty$.

Execution time constraints. For a given resource $r \in R$ these constraints conjunct guards of actions issued from states $s \in U_{i,r}$, in particular end actions. They are usually constraints of the form $E_{i,r}^l \leq x_{i,r} \leq E_{i,r}^u$ where $E_{i,r}^l > 0$ and $E_{i,r}^u$ are respectively the lower and the upper bounds of the execution time of P_i on r.

Deadline constraints. These constraints express the fact that since the process arrival a resource r has been used within a given deadline $D_{i,r}$. They are usually X-constraints of the form $t_i \leqslant D_{i,r}$ constraining the end transitions on r. The usual notion of deadline of a process is thus the deadline of the process on the processor.

Urgency. Begin actions are usually eager to denote the fact that a process takes the resource as soon as it is allowed to do so. End actions are delayable, meaning that the action may arrive at any time, but before the action becomes disabled forever. Arrival and request actions are in general delayable for the same reason; arrival actions of a sporadic process are lazy, expressing the fact that there is no deadline on the process arrival.

Example 3.4 *Coming back to the process* `P` *of example 2.2, clock t_p measures the inter-arrival time, which is specified by the guard of transition* arrive *to be at least 5 time units. Hence, P is a sporadic process. The execution time on resource CPU is measured by clock x_p, which is reset at the only action entering $U_{P,CPU} = \{computing\}$, that is,* compute. *The execution time is specified by the guard of transition* write *to be within 2 and 3 time units; the same guard requires as a deadline constraint that the execution be finished before t_p reaches the minimal inter-arrival time.*

Transition arrive *is lazy, meaning that there is no deadline on the arrival of P. Transition* compute *is eager, specifying that the process is eager to acquire the resource. Transition* write *is delayable, guaranteeing it to be taken before it becomes disabled, in spite of its uncontrollable character.*

Table 3.1 summarizes the classification of process actions with corresponding timing constraints.

action type	ctrl.	most liberal guard	urgency	resets
arrival of process P_i	**u**	$T_i^l \leqslant t_i \leqslant T_i^u$	δ	t_i
begin of P_i on r	**c**	$t_i \leqslant D_{i,r} - E_{i,r}^u$	ϵ	$x_{i,r}$
end of P_i on r	**u**	$E_{i,r}^l \leqslant x_{i,r} \leqslant E_{i,r}^u \ \wedge\ t_i \leqslant D_{i,r}$	δ	$\emptyset$
timeout of P_i on r	**u**	$t_i > D_{i,r} - E_{i,r}^u$	ϵ	$\emptyset$

Table 3.1: Action classification.

A system of cooperating processes is specified by a set of processes P_i modelled by timed systems TS_i, a set of resources R_p and R_n, and the associated predicates $U_{i,r}$.

Multiple instances of a process

Within our modelling framework, multiple instances of a cyclic process can be taken into account if the number of instances existing at the same time is bounded, say, by n. This has been shown to be decidable in [EWY99] for the non-preemptive case.

The process is then modelled by a cyclic *starter* process, and n *instances* of the process to be modelled, whose arrival actions are controllable, guarded by *true*, and synchronized with the arrival action of the starter process. Any time the starter process executes an arrival action, there is some instance process that is sleeping and ready to synchronize, according to the hypothesis of a bounded number of co-existing instances.

3.2 Process Interaction

The synchronization layer ensures mutual exclusion on shared resources, and provides a preemption mechanism. It is in charge of guaranteeing only functional properties of the application.

The synchronization layer is modelled in an incremental manner. First, a general preemption mechanism is provided that allows any process acquiring a preemptable resource to preempt any other process using that resource. This is done by synchronizing the processes with timed systems modelling the states of the preemptable resources. Inter-process communication other than preemption, is provided by synchronizing communicating transitions of two or more processes.

Then, the behaviour of the synchronizing processes is restricted by a priority function ensuring mutual exclusion on the use of the shared resources. If desired, sequences of actions can be made atomic by another priority function. We define priority functions ensuring the above properties, which can be combined, similar to building blocks of a toolbox, to obtain the synchronization layer of the real-time application to be modelled.

3.2.1 Modelling with Priorities

Efforts to reduce the cost, accelerate the development, and improve maintainability of embedded systems target at partitioning [Rus99], that is, the integration of distinct processes — which may have different levels of criticality — on one hardware platform. In such systems, it is particularly important to guarantee the absence of unwanted interference between the processes, which make the system behaviour hard or impossible to predict. In the context of distributed systems, the problem of different functionalities that turn out to be inconsistent, is known as *feature interaction* (see for example [dB99]).

Priority functions are a natural and intuitive means for expressing functional as well as non-functional aspects of process interaction such as mutual exclusion, atomicity, or scheduling policies. Moreover, they provide diagnostics allowing to detect inconsistencies between different aspects of coordination between processes at an early design stage: as discussed earlier, the sum $\oplus$ of priority functions is undefined for system states having a circuit in the union of the associated priority orders, indicating the risk of deadlock. Conversely, the property of activity preservation guarantees that a well-defined priority function does not introduce deadlocks in a timed system.

Generally, a circuit is due to contradictory priority functions. This is an important argument in favour of modelling and integrating as many interactional aspects as possible by priority functions, which allows to detect inconsistencies within some level of interaction — for example, inter-process synchronization, or conflict resolution by a scheduler — as well as between different levels of interaction. For example, modelling both atomicity of action sequences, and resource allocation under the priority ceiling protocol in the formalism of priority functions, allows to integrate both policies in one formal description, and to detect possible inconsistencies.

The diagnostic comprises the set of system states, consisting of control states and clock valuations for which the problem appears, and of the set of conflicting actions. Control states for which the priority function to be applied is undefined for some clock valuation, are considered as error states and must be made unreachable by some priority function, or using the priority synthesis algorithm presented in section 4.3.1.

Once a safety property $\mathcal{P}$ has been modelled with some priority rule pr, property 2.4 makes sure that $\mathcal{P}$ is still guaranteed by the sum of pr and further priority rules. This composability principle, which has been studied in [AGS00], is the key to a modular description of different behavioural aspects by a set of priority functions. Their composition provides a priority function integrating the different aspects of process interaction. This composability result suggests an incremental design process, where the model of the real-time application is progressively restricted until the system satisfies all properties that can be expressed as priority functions. If an inconsistency is detected, it can be backtracked up to a set of contradictory properties.

3.2.2 Preemption

Preemption (see for example [Ber93]) allows to suspend a process using some resource, in order to grant the resource to an arriving process whose execution is, at that moment, considered of higher importance. Preemption allows to use resources more efficiently, and to apply simpler scheduling strategies than in the non-preemptive case [GMR95, GRS96].

Definition 3.1 ($\bullet$ and $\circ$) *Let $TS = (S, A, T, X, \mathbf{b}, h)$ be a timed system, and $M \subseteq S$. Adopting a Petri-net like notation, we define the set of actions entering M and issued from M as*

$$\begin{aligned} \bullet M &= \{a \in A \mid \exists (s, a, s') \in T \, . \, s \notin M \ \wedge \ s' \in M\} \quad \textit{and} \\ M \circ &= \{a \in A \mid \exists (s, a, s') \in T \, . \, s \in M\} \, , \end{aligned}$$

respectively.

To model preemption mechanisms, we use for each process P_i and each preemptable resource $r \in R_p$ a timed system $PMTN_{i,r}$ (see figure 3.3) differentiating states of P_i using r according to whether P_i effectively uses r, or is preempted from r. $PMTN_{i,r}$ has three control states $n\_use_{i,r}$, $active_{i,r}$ and $susp_{i,r}$ meaning respectively that the process P_i is not using r, is effectively using r, is suspended due to preemption from r. The synchronization of $PMTN_{i,r}$ and TS_i is defined so as to agree with this meaning, that is, TS_i is at a state s such that $\neg U_{i,r}(s)$ if and only if $PMTN_{i,r}$ is at state $n\_use_{i,r}$.

Following notation from semaphores, the actions $p_{i,r}$ and $v_{i,r}$ represent granting r to P_i and freeing of r by P_i, respectively. The actions $pt_{i,r}$ and $rs_{i,r}$ represent P_i getting preempted from r and resuming r, respectively. Since the execution time of P_i on r is bounded by $E^u_{i,r}$, P_i may be preempted from r, and resume its execution on r, as long as $x_{i,r} \leqslant E^u_{i,r}$. The preemption transition is lazy, since the specification of P_i does not require it to be taken. The resuming transition $rs_{i,r}$ is eager, meaning that P_i takes r as soon as it is allowed to. The transitions $p_{i,r}$ and $v_{i,r}$ are lazy, which means that any transition synchronizing with them keeps its own urgency type in the synchronization transition, according to definition 2.13.

When a process P_i is suspended due to preemption from some resource r, all the execution clocks of TS_i are stopped: $\forall r \in R \, . \, \mathbf{b}_{susp_{i,r}}[x_{i,r}] = 0$.

The synchronization between TS_i and the timed systems $PMTN_{i,r}$ is specified by the set of synchronization actions Σ_i:

$$\begin{aligned} \Sigma_i \quad = \quad & \{b_i | p_{i,r} \mid r \in R_p \ \wedge \ b_i \in BGN_{i,r}\} \ \cup \\ & \{e_i | \{v_{i,r} \mid r \in R_p \ \wedge \ e_i \in END_{i,r}\} \mid \exists r \in R_p \, . \, e_i \in END_{i,r}\} \, . \end{aligned}$$

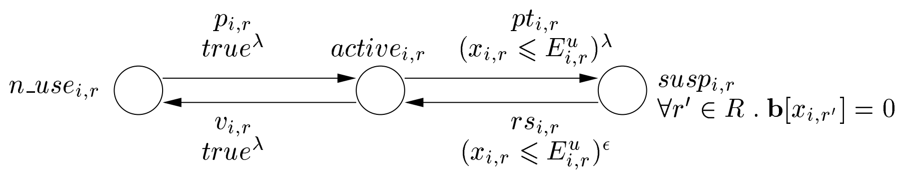


Figure 3.3: The preemption management system $PMTN_{i,r}$.

Intuitively, each begin action b_i of process P_i acquiring a preemptable resource r synchronizes with an allocating action $p_{i,r}$. End actions $end_{i,r}$ synchronize with freeing actions v_{i,r_j} of $PMTN_{i,r_j}$ for all preemptable resources r_j they free.

Tight synchronization between TS_i and $PMTN_{i,r}$ is achieved by restricting the product timed system with the following action constraint, which prevents synchronizing actions of $PMTN_{i,r}$ from interleaving, and TS_i from progressing while it is preempted from some resource:

$$K_i = \bigcup_{r \in R_p} \{(n\_use_{i,r}, \{p_{i,r}\}),\ (active_{i,r}, \{v_{i,r}\}),$$
$$(susp_{i,r}, \{a_i,\ a_i \shortmid p_{i,r'} \mid a_i \in U_{i,r} \circ\ \wedge r' \in R_p\})\}\ .$$

Preemption transitions $pt_{i,r}$ must be prevented from interleaving, but they must not be cut off by an action constraint, since they are needed for synchronization with the timed systems of other processes. Therefore, they are disabled by the priority function

$$pr'_i \;=\; \bigoplus_{r \in R_p} \{active_{i,r},\ pt_{i,r} \prec_{\infty} (U_{i,r} \circ \cup \{rs_{i,r'} \mid r' \in R_p\})\}$$

preventing pt-actions from interleaving by disabling them whenever some other action of TS_i, or a resuming action, will eventually become possible. For a given timed system TS_i, let

$$(TS_i^{PMTN},\ pr_{\Sigma,i}) \;=\; \left(\|_{\Sigma_i}\{TS_i\} \cup \{PMTN_{i,r} \mid r \in R_p\ \wedge\ U_{i,r} \neq \emptyset\}\right) \,/\, K_i$$

be the timed system representing TS_i put in parallel with the preemption automata, and let

$$(TS_i^{PMTN},\ pr_i) \;=\; (TS_i^{PMTN},\ pr_{\Sigma,i} \oplus pr'_i)$$

be the same timed system to which pr'_i has been applied. This construction amounts to adding, for each using state, a preempted state that is reached by a preempting action, and that can be left by a resuming action, returning to the using state.

Example 3.5 *The timed system* TS_1 *of figure 3.4 models a process using two preemptable resources,* r_1 *and* r_2*, according to* $U_{1,r_1} = \{u_1, u_{1,2}\}$*, and* $U_{1,r_2} = \{u_{1,2}\}$*. For the sake of simplicity, timing information has been omitted.*

Preemptability of the resources is modelled by synchronizing TS_1 *with two timed systems* $PMTN_{1,r_1}$ *and* $PMTN_{1,r_2}$*. Figure 3.5 shows the timed system* $\|_{\Sigma_1}\{TS_1,$

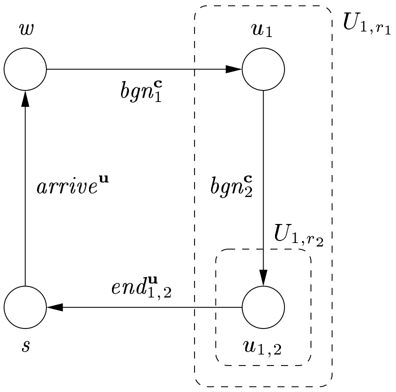


Figure 3.4: TS_1 using two preemptable resources r_1 and r_2.

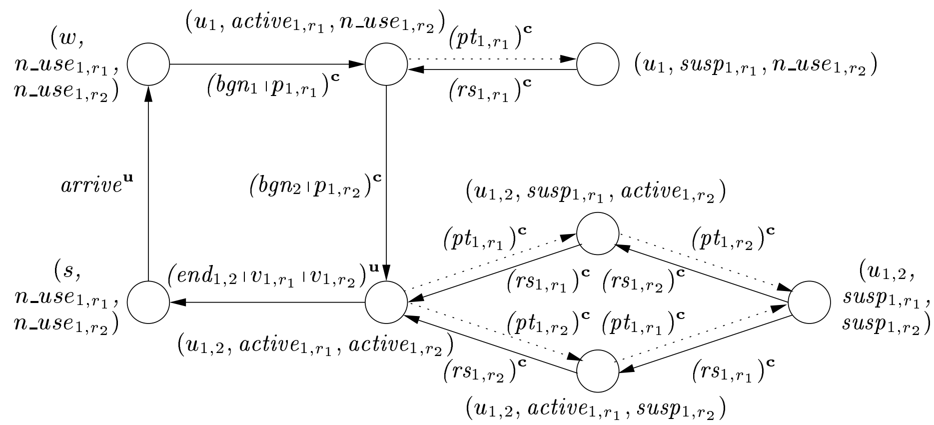


Figure 3.5: Product of TS_1, $PMTN_{1,r_1}$ and $PMTN_{1,r_2}$, modelling preemption of TS_1 from r_1 and r_2. Dotted transitions are disabled by the priority function pr'_1.

$PMTN_{1,r_1}, PMTN_{1,r_2}\}$. Dotted transitions synchronize with other timed systems TS_i, expressing TS_i preempting TS_1, but they are prevented from interleaving by the priority function pr'_1.

Preemption between the different processes is modelled by parallel composition of the timed systems TS_i^{PMTN} with the synchronization actions Σ_{PMTN}:

$$\Sigma_{PMTN} = \{b_i | p_{i,r} | pt_{j,r} \mid i, j \in \{1, \dots, n\} \ \wedge \ i \neq j \ \wedge \ r \in R_p \ \wedge \ b_i \in BGN_{i,r}\} \ .$$

Intuitively, each begin action b_i of some process P_i acquiring a preemptable resource r synchronizes with an allocating action $p_{i,r}$ and, if there is already a process using r, with the preempting action $pt_{j,r}$. The maximal progress paradigm makes sure that the latter action does effectively participate in the synchronization if r is not free.

The timed system of processes with preemption is obtained as

$$(TS, pr) \;=\; \|_{\Sigma_{PMTN}} \big\{ (TS_i^{PMTN},\; pr_i) \mid i \in \{1, \ldots, n\} \big\} \;.$$

The property that $PMTN_{i,r}$ is in state $n\_use_{i,r}$ if and only if P_i is not in $U_{i,r}$, is an inductive invariant: the actions of $PMTN_{i,r}$ entering and leaving $n\_use_{i,r}$, that is, $p_{i,r}$ and $v_{i,r}$, do not interleave, but always synchronize with the $BGN_{i,r}$ and $END_{i,r}$ actions of TS_i. The actions in $U_{i,r}\circ$ are disabled by the action constraint K_i as long as P_i is preempted from r. Therefore, there is a tight synchronization between TS_i and $PMTN_{i,r}$ on the actions entering and leaving $U_{i,r}$ and $active_{i,r} \vee susp_{i,r}$, respectively.

The timed system TS_i^{PMTN} presents a refined view of TS_i, in the sense that it allows to distinguish, when TS_i is in the using state of a preemptable resource, whether TS_i is effectively executing the resource, or preempted from it. This more detailed view motivates the definition of using and waiting states of the timed system TS_i^{PMTN} in the following way. For any control state s of TS_i^{PMTN}, let

$$U'_{i,r}(s) \iff U_{i,r}(s) \;\wedge\; \big(r \in R_n \vee active_{i,r}(s)\big) \;.$$

For a non-preemptable resource $r \in R_n$, $U'_{i,r}$ and $U_{i,r}$ are identical. The predicate $W'_{i,r}$ can be derived from $U'_{i,r}$ according to its definition: $W'_{i,r}$ characterizes the states that are not in $U'_{i,r}$, but from which there is a transition of TS_i^{PMTN} leading to some state in $U'_{i,r}$. It follows that $W'_{i,r} \iff W_{i,r} \vee (r \in R_p \;\wedge\; susp_{i,r})$. In the same way, we define the set $BGN'_{i,r}$ of actions of TS_i^{PMTN} entering the set of using states, as $BGN'_{i,r} = \bullet U'_{i,r}$. That is, $BGN'_{i,r} = BGN_{i,r}$ for $r \in R_n$, according to the hypothesis that actions acquiring a non-preemptable resource do not acquire a preemptable resource at the same time, and that they do not synchronize. For $r \in R_p$, we have $BGN'_{i,r} = \{b_i \shortmid p_{i,r},\; rs_{i,r} \mid b_i \in BGN_{i,r}\}$. Notice that preempting actions $b_i \shortmid p_{i,r} \shortmid pt_{j,r}$ for $b_i \in BGN_{i,r}$ are not in $BGN'_{i,r}$, since they do not belong to the set of actions of the component TS_i^{PMTN}.

This methodology does not take context switching times into account. However, they can be expressed by slightly modifying the model, for example by adding an intermediate state $latent_{i,r}$ in which some time must pass after a $rs_{i,r}$ action, before $PMTN_{i,r}$ gets back to $active_{i,r}$.

3.2.3 Communication

In order to express inter-process communication, we allow actions in $A_i \setminus (\bigcup_{r \in R} BGN_{i,r} \cup \bigcup_{r \in R_p} U_{i,r}\circ)$ of the timed systems $TS_1, \ldots, TS_n$ to synchronize. Let Σ be the set of synchronization actions. Notice that actions issued from a state using a preemptable resource could be allowed to synchronize, provided that the action constraint K above is modified so that it disables not only interleaving, but also synchronization actions issued from a preempted state.

3.2.4 Mutual Exclusion

Consider a system of cooperating processes sharing common resources. The goal of this section is to restrict the behaviour of the processes so that the global behaviour satisfies mutual exclusion on the use of the shared resources. We propose a method based on priorities to model mutual exclusion.

Consider two processes P_1 and P_2 modelled by the timed systems $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$, $i \in \{1,2\}$, and critical sections $M_1 \subseteq S_1$ and $M_2 \subseteq S_2$ that are intended to be in mutual exclusion — written $M_1 \# M_2$ —, as depicted in figure 3.6. We have

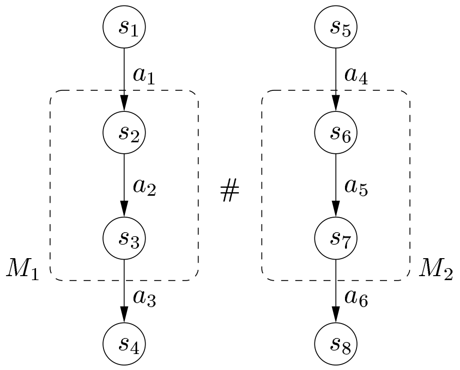


Figure 3.6: Mutual exclusion between M_1 and M_2.

$\bullet M_1 = \{a_1\}$, $M_1\circ = \{a_2, a_3\}$, $\bullet M_2 = \{a_4\}$, and $M_2\circ = \{a_5, a_6\}$. Mutual exclusion between M_1 and M_2 means that whenever TS_1 is ready to enter M_1, and TS_2 is already in M_2, then TS_2 must leave M_2 before TS_1 can enter M_1, and vice versa. This order is expressed by the priority order $\bullet M_1 \prec_\infty M_2\circ$, applied to states where P_1 is waiting to enter M_1 and P_2 is in M_2, and $\bullet M_2 \prec_\infty M_1\circ$ when P_1 is in M_1, and P_2 is waiting to enter M_2. Formally,

$$pr = \left\{ \left(s_1 \wedge (s_6 \vee s_7),\ \bullet M_1 \prec_\infty M_2 \circ \right),\ \left(s_5 \wedge (s_2 \vee s_3),\ \bullet M_2 \prec_\infty M_1 \circ \right) \right\} .$$

As long as none of the timed systems gets deadlocked in its critical section, mutual exclusion is guaranteed by the priority order *pr*:

Theorem 3.1 (Mutual exclusion) *Let $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$, $i \in \{1,2\}$, be two timed systems, $M_1 \subseteq S_1$ and $M_2 \subseteq S_2$ be predicates characterizing the states of two critical sections, and before (M_i) characterize the states in $S_i \setminus M_i$ from which a transition leads into M_i, for $i \in \{1,2\}$. Then, the priority function*

$$pr = \left\{ \left(\mathit{before}\ (M_1) \wedge M_2,\ \bullet M_1 \prec_\infty M_2 \circ \right),\ \left(M_1 \wedge \mathit{before}\ (M_2),\ \bullet M_2 \prec_\infty M_1 \circ \right) \right\}$$

ensures mutual exclusion between M_1 and M_2 in the timed system $TS = \|_\emptyset \{TS_1, TS_2\}$ as long as none of the component systems gets deadlocked in its critical section.

Proof. We show that from a control state $s = (s_1, s_2)$ of TS respecting the mutual exclusion, no transition issued from s leading to a state violating the mutual exclusion is enabled. This is true if none of the processes is in the critical section in s, since TS is the asynchronous product without synchronizing actions.

Suppose that exactly one of the timed systems is in its critical section, say, $M_1(s_1)$, and the other one is ready to enter: $(\mathit{before}\ (M_2))(s_2)$. Any "dangerous" transition $t = ((s_1, s_2), a, (s_1, s_2')) \in T$ with $a \in A_2$ and $M_2(s_2')$ leading TS_2 into M_2 is labelled by an action in $\bullet M_2$. Let the set $\mathcal{I}$ enumerate the transitions $t_i = ((s_1, s_2), a_i, (s_1', s_2))$ with $a_i \in A_1$, and suppose that $h(t_i) = (g_i, \tau_i, r_i)$. The actions a_i are elements of $M_1\circ$. If $h(t) = (g, \tau, r)$, then the guard of the "dangerous" transition t is restricted by pr to $g' = g \wedge \neg \bigvee_{i \in \mathcal{I}} \Diamond^{\mathbf{b}_s} g_i$ in the timed systems with priorities (TS, pr). Consider some clock valuation $\mathbf{x} \in \mathbb{D}^m$. The condition that TS_1 is not deadlocked in $(s, \mathbf{x})$ means that some transition leaving s will eventually become enabled, that is, $\exists d \in \mathbb{D}_{>0}) \ . \ \bigvee_{i \in \mathcal{I}} g_i(\mathbf{x} + d\mathbf{b}_s)$. Therefore, $\big(\bigvee_{i \in \mathcal{I}} \Diamond^{\mathbf{b}_s} g_i\big)(\mathbf{x})$, which implies that $\neg g'(\mathbf{x})$, which means that t is disabled. ∎

Notice that the above construction restricts the untimed transition structure in a minimal manner, that is, only transitions leading from a state satisfying mutual exclusion to a state violating mutual exclusion, are restricted. A sufficient condition ensuring that none of the component systems gets deadlocked in a critical section, is introduced in chapter 4 with the property of *structural safety*, which can be guaranteed by construction.

Applied to the set of states using some resource $r \in R$, the priority function ensuring mutual exclusion between the timed systems $TS_1, \ldots, TS_n$ becomes

$$pr^r_{\mathsf{mutex}} = \bigoplus_{i,j \in \{1,\ldots,n\}} \{(U'_{i,r} \wedge W'_{j,r},\ BGN^*_{j,r} \prec_\infty U'_{i,r}\circ)\}\ ,$$

where $BGN^*_{j,r}$ denotes the set of actions allocating r to TS_j when r is free: $BGN^*_{j,r} = BGN_{j,r}\ \cup\ \{b_j \mid pt_{i,r'} \mid b_j \in BGN'_{j,r}\ \wedge\ i \in \{1,\ldots,n\} \setminus \{j\}\ \wedge\ r' \in R_p \setminus \{r\}\}$. Intuitively, in states of the product timed system where P_i is using r and P_j is waiting for r, actions of P_i have unbounded priority over any action allocating r to P_j. Notice that the sets of states satisfying the predicate $U'_{i,r} \wedge W'_{j,r}$ for different indices i and j may not be disjoint — for example, a state can be a waiting state for several processes.

Let $pr_{\mathsf{mutex}} = \bigoplus_{r \in R} pr^r_{\mathsf{mutex}}$, and

$$(TS, pr)\ =\ \|_{\Sigma_{PMTN} \cup \Sigma} \{(TS_i^{PMTN},\ pr_i) \mid i \in \{1,\ldots,n\}\}$$

be the timed system of processes constructed so far.

Example 3.6 *The two timed systems TS_1 and TS_2 shown in figure 3.7 model processes sharing two non-preemptable resources r_1 and r_2. TS_1 acquires first r_1 and then r_2, whereas TS_2 proceeds in the inverse order. Mutual exclusion is modelled by*

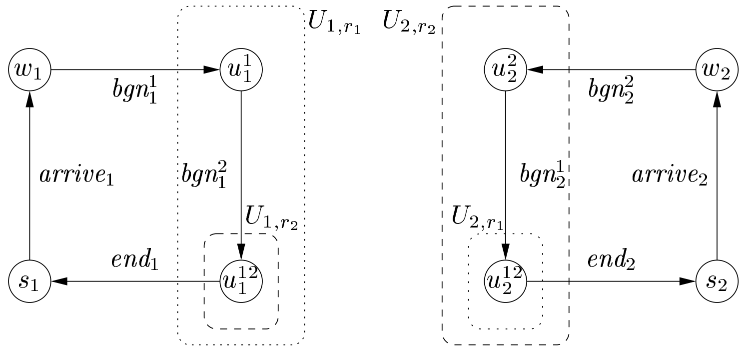


Figure 3.7: Two processes sharing two non-preemptable resources.

applying the priority functions ensuring mutual exclusion on each of the two resources:

$$
\begin{aligned}
pr_{\mathsf{mutex}} &= pr^{r_1}_{\mathsf{mutex}} \oplus pr^{r_2}_{\mathsf{mutex}} \\
&= \big\{\big(U_{1,r_1} \wedge W_{2,r_1},\ BGN_{2,r_1} \prec_\infty U_{1,r_1}\circ\big),\ \big(U_{2,r_1} \wedge W_{1,r_1},\ BGN_{1,r_1} \prec_\infty U_{2,r_1}\circ\big)\big\} \oplus \\
&\quad\ \big\{\big(U_{1,r_2} \wedge W_{2,r_2},\ BGN_{2,r_2} \prec_\infty U_{1,r_2}\circ\big),\ \big(U_{2,r_2} \wedge W_{1,r_2},\ BGN_{1,r_2} \prec_\infty U_{2,r_2}\circ\big)\big\} \\
&= \big\{\big((u_1^1 \vee u_1^{12}) \wedge u_2^2,\ bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}\big),\ \big(u_2^{12} \wedge w_1,\ bgn_1^1 \prec_\infty end_2\}\big)\big\} \oplus \\
&\quad\ \big\{\big(u_1^{12} \wedge w_2,\ bgn_2^2 \prec_\infty end_1\}\big),\ \big((u_2^2 \vee u_2^{12}) \wedge u_1^1,\ bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}\big)\big\} \\
&= \big\{\big(u_1^1 \wedge u_2^2,\ (bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}) \oplus (bgn_1^2 \prec_\infty \{bgn_2^1, end_2\})\big), \\
&\quad\ \big(u_1^{12} \wedge u_2^2,\ bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}\big),\ \big(u_2^{12} \wedge w_1,\ bgn_1^1 \prec_\infty end_2\}\big), \\
&\quad\ \big(u_1^{12} \wedge w_2,\ bgn_2^2 \prec_\infty end_1\}\big),\ \big(u_2^{12} \wedge u_1^1,\ bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}\big)\big\} \\
&= \big\{\big(u_1^1 \wedge u_2^2,\ \bot\big),\ \big(u_1^{12} \wedge u_2^2,\ bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}\big),\ \big(u_2^{12} \wedge w_1,\ bgn_1^1 \prec_\infty end_2\}\big), \\
&\quad\ \big(u_1^{12} \wedge w_2,\ bgn_2^2 \prec_\infty end_1\}\big),\ \big(u_2^{12} \wedge u_1^1,\ bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}\big)\big\}
\end{aligned}
$$

Consider the sum $(bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}) \oplus (bgn_1^2 \prec_\infty \{bgn_2^1, end_2\})$ *above, evaluated to* $\bot$*: it is not a priority order, since the union of the priority orders of both arguments contains* $bgn_2^1 \prec_\infty bgn_1^2$ *and* $bgn_1^2 \prec_\infty bgn_2^1$*. This suggests that the specification may be deadlocking in the states where the priority circuit occurs, that is,* $u_1^1 \wedge u_2^2$*. Indeed, if* TS_1 *has acquired* r_1 *and* TS_2 *has acquired* r_2*, then both processes are in a deadlock situation, each one waiting for the other process to free the resource it is holding. This well-known example illustrates how our modelling framework provides diagnostics for contradictory or potentially deadlocking specifications.*

Theorem 3.2 (Mutual exclusion) *The priority function* pr_{mutex} *ensures on TS mutual exclusion between states using some resource* $r \in R$*, that is, between*

1. $U_{i,r}$ *and* $U_{j,r}$ *for* $r \in R_n$ *and* $i \neq j$*, and*

2. $active_{i,r}$ *and* $active_{j,r}$ *for* $r \in R_p$ *and* $i \neq j$*,*

as long as none of the processes gets deadlocked while holding a resource.

Proof. 1. The proof is similar to that of 3.1: since $BGN'_{i,r} = BGN_{i,r}$ and $U'_{i,r} = U_{i,r}$, pr_{mutex} implies that $BGN^*_{i,r} \prec_\infty U_{j,r}\circ$ for states s with $W_{i,r}(s) \wedge U_{j,r}(s)$. Actions in $BGN^*_{i,r}$ are thus disabled by actions in $U_{i,r}\circ$ according to pr_{mutex}, as long as TS_i does not get deadlocked in $U_{i,r}$.

2. It has been shown that $PMTN_{i,r}$ is in state $n\_use_{i,r}$ if and only if P_i is not in $U_{i,r}$. Actions entering $active_{i,r}$ are $b_{i,r} \mid p_{i,r}$, $b_{i,r} \mid p_{i,r} \mid pt_{j,r}$ for $b_{i,r} \in BGN_{i,r}$ and $j \neq i$, and $rs_{i,r}$. pr_{mutex} ensures that $b_{i,r} \mid p_{i,r}$ and $rs_{i,r}$ are disabled by possible actions in $U'_{j,r}\circ$. Finally, action $b_{i,r} \mid p_{i,r} \mid pt_{j,r}$ preempting process P_j keeps the number of processes active on r constant, that is, equal to one. ■

Example 3.7 (Mutual exclusion with preemption) *Consider three cyclic processes P_1, P_2, and P_3 sharing one preemptable processor (CPU). P_1 and P_3 share a non-preemptable resource r. Figure 3.8 shows the timed systems TS_i^{PMTN}, $i \in \{1,2,3\}$, modelling the three processes, which are already synchronized with the preemption timed systems $PMTN_{i,CPU}$, according to the methodology of section 3.2.2. In order to sim-*

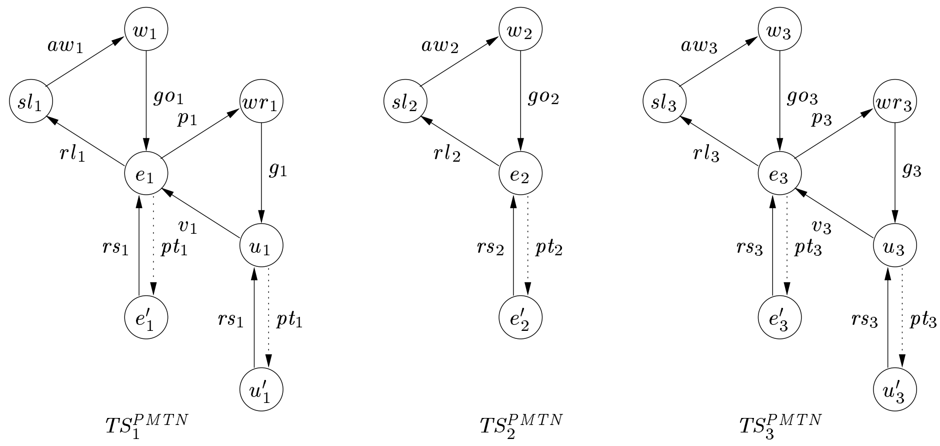


Figure 3.8: Three processes sharing a *CPU* and a resource.

plify the figure and the following text, the states and actions of the timed systems have been renamed. For example, preemption and resuming actions are called pt_i and rs_i instead of $pt_{i,CPU}$ and $rs_{i,CPU}$, respectively, since CPU is the only preemptable resource. Preempted states — where the component $PMTN_{i,CPU}$ is in state $susp_{i,CPU}$ — are primed in the figure. Timing information has been omitted in this example. In state sl_i, w_i, e_i, wr_i, u_i, process P_i is sleeping, waiting for the CPU, running on the CPU, waiting for the resource r, and using r, respectively. The actions aw_i, go_i, p_i, g_i, v_i, and rl_i of process P_i stand for awake, begin execution, request resource, get resource, free resource, and end of execution, respectively. The primed states and the rs_i and synchronizing pt_i actions model preemption. The following table summarizes *the use of the CPU and resource r.*

$$
\begin{array}{rlrlrl}
U_{1,CPU} &= \{e_1, u_1\} & U_{2,CPU} &= \{e_2\} & U_{3,CPU} &= \{e_3, u_3\} \\
BGN_{1,CPU} &= \{go_1, g_1, rs_1\} & BGN_{2,CPU} &= \{go_2, rs_2\} & BGN_{3,CPU} &= \{go_3, g_3, rs_3\} \\
U_{1,CPU^\circ} &= \{p_1, v_1, rl_1\} & U_{2,CPU^\circ} &= \{rl_2\} & U_{3,CPU^\circ} &= \{p_3, v_3, rl_3\} \\
W_{1,CPU} &= \{w_1, e_1', wr_1, u_1'\} & W_{2,CPU} &= \{w_2, e_2'\} & W_{3,CPU} &= \{w_3, e_3', wr_3, u_3'\} \\
U_{1,r} &= \{u_1, u_1'\} & U_{2,r} &= \emptyset & U_{3,r} &= \{u_3, u_3'\} \\
BGN_{1,r} &= \{g_1\} & BGN_{2,r} &= \emptyset & BGN_{3,r} &= \{g_3\} \\
U_{1,r^\circ} &= \{rs_1, v_1\} & U_{2,r^\circ} &= \emptyset & U_{3,r^\circ} &= \{rs_3, v_3\} \\
W_{1,r} &= \{wr_1\} & W_{2,r} &= \emptyset & W_{3,r} &= \{wr_3\}
\end{array}
$$

Preemption is modelled by the set of synchronization actions $\Sigma_{PMTN} = \{go_1 \mid pt_2, go_1 \mid pt_3, g_1 \mid pt_2, g_1 \mid pt_3, go_2 \mid pt_1, go_2 \mid pt_3, go_3 \mid pt_1, go_3 \mid pt_2, g_3 \mid pt_1, g_3 \mid pt_2\}$. *Intuitively, each action in* $BGN_{i,CPU}$ *entering the set of states using the CPU synchronizes with the preemption action* pt_j *of any other process. The timed systems* TS_i^{PMTN} *are prevented in states* $s \in U_{i,CPU}$ *from executing spontaneously a preemption action, by the priorities* $pr_i' = \{(active_{i,CPU},\ pt_i \prec_\infty U_{i,CPU}\circ)\}$. *For* $i \in \{1,3\}$, *we obtain* $pr_i' = \{(e_i \vee u_i,\ \{pt_i \prec_\infty rl_i,\ pt_i \prec_\infty p_i,\ pt_i \prec_\infty v_i\})\}$, *and* $pr_2' = \{(e_i,\ \{pt_i \prec_\infty rl_i\})\}$.

Mutual exclusion is modelled by the priority function $pr_{\mathsf{mutex}} = pr_{\mathsf{mutex}}^{CPU} \oplus pr_{\mathsf{mutex}}^{r}$, *where*

$$
\begin{aligned}
pr_{\mathsf{mutex}}^{CPU} &= \bigoplus_{i,j \in \{1,2,3\}} \{(U_{i,CPU} \wedge W_{j,CPU},\ BGN_{j,CPU}^{*} \prec_\infty U_{i,CPU^\circ})\} \\
&= \{((e_1 \vee u_1) \wedge (w_2 \vee e_2'),\ \{go_2, rs_2\} \prec_\infty \{p_1, v_1, rl_1\}), \\
&\qquad (e_2 \wedge (w_1 \vee e_1' \vee wr_1 \vee u_1'),\ \{go_1, g_1, rs_1\} \prec_\infty rl_2)\} \oplus \\
&\qquad \{((e_1 \vee u_1) \wedge (w_3 \vee e_3' \vee wr_3 \vee u_3'),\ \{go_3, g_3, rs_3\} \prec_\infty \{p_1, v_1, rl_1\}), \\
&\qquad ((e_3 \vee u_3) \wedge (w_1 \vee e_1' \vee wr_1 \vee u_1'),\ \{go_1, g_1, rs_1\} \prec_\infty \{p_1, v_1, rl_1\})\} \oplus \\
&\qquad \{(e_2 \wedge (w_3 \vee e_3' \vee wr_3 \vee u_3'),\ \{go_3, g_3, rs_3\} \prec_\infty rl_2), \\
&\qquad ((e_3 \vee u_3) \wedge (w_1 \vee e_1' \vee wr_1 \vee u_1'),\ \{go_2, rs_2\} \prec_\infty \{p_3, v_3, rl_3\})\} \\
pr_{\mathsf{mutex}}^{r} &= \bigoplus_{i,j \in \{1,2,3\}} \{(U_{i,r} \wedge W_{j,r},\ BGN_{j,r}^{*} \prec_\infty U_{i,r^\circ})\} \\
&= \{((u_1 \vee u_1') \wedge wr_3,\ \{g_3, g_3 \mid pt_1, g_3 \mid pt_2\} \prec_\infty \{v_1, rs_1\}), \\
&\qquad ((u_3 \vee u_3') \wedge wr_1,\ \{g_1, g_1 \mid pt_2, g_1 \mid pt_3\} \prec_\infty \{v_1, rs_1\})\}\ .
\end{aligned}
$$

The synchronized system is described by

$$
\left(\|_{\Sigma_{PMTN}} \{(TS_i^{PMTN}, pr_i') \mid i \in \{1,2,3\}\},\ pr_{\mathsf{mutex}}\right)\ .
$$

We will come back to this example when dealing with scheduling policies.

3.2.5 Atomicity

In order to allow the expression of atomicity of action sequences with respect to a set of actions, we use *transient* states, a refinement of the concept of [BFG$^+$00]. Consider the set of timed systems $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$, $i \in \{1,\dots,n\}$. For a given $i \in \{1,\dots,n\}$, a set of states $S^i \subseteq S_i$ is transient with respect to a set of controllable actions $A^i \subseteq \bigcup_{j\neq i} A_j^c$ if all actions in A^i are disabled as long as TS_i stays in S^i. We model this atomicity constraint by the priority function

$$pr^i_{\mathsf{atom}} = \big\{\big(S^i,\ A^i \prec_\infty S^i\circ\big)\big\} \oplus \bigoplus_{r\in R_p} \big\{\big(S^i \wedge susp_{i,r},\ A^i \prec_\infty rs_{i,r}\big)\big\}\ .$$

Intuitively, actions in A^i are disabled by actions issued from S^i, and by resuming actions $rs_{i,r}$ when TS_i is preempted while in some state of S^i.

The atomicity constraints to be ensured on the set of timed systems are expressed by $pr_{\mathsf{atom}} = \bigoplus_{i\in\{1,\dots,n\}} pr^i_{\mathsf{atom}}$. Let (TS, pr) be the product timed system constructed so far.

Theorem 3.3 (Atomicity) *The priority function pr_{atom} ensures that for any $i \in \{1,\dots,n\}$, S^i is transient with respect to the actions in A^i in the timed system $\big(TS, pr \oplus pr_{atom}\big)$, as long as TS_i does not get deadlocked in S^i.*

Proof. In states where P_i is not preempted, the actions in A^i are disabled by pr^i_{atom}; the proof is very similar to that of theorem 3.1. In states where P_i is preempted, say, from resource $r \in R_p$, pr^i_{atom} ensures that $A_i \prec_\infty rs_{i,r}$. The guard of $rs_{i,r}$ is *true*, such that the actions in A_i are disabled by $rs_{i,r}$. ∎

Example 3.8 *In order to make the deadlocking state (u_1^1, u_2^2) of example 3.6 unreachable, one can choose to make the state u_1^1 (resp. u_2^2) transient with respect to the action bgn_2^2 (resp. bgn_1^1) allocating the second resource. This atomicity constraint is expressed by the priority function*

$$\begin{aligned}
pr_{atom} &= pr^2_{atom} \oplus pr^3_{atom}\\
&= \big\{\big(u_1^1,\ bgn_2^2 \prec_\infty \{u_1^1\}\circ\big)\big\} \oplus \big\{\big(u_2^2,\ bgn_1^1 \prec_\infty \{u_2^2\}\circ\big)\big\}\\
&= \big\{\big(u_1^1,\ bgn_2^2 \prec_\infty bgn_1^2\big)\big\} \oplus \big\{\big(u_2^2,\ bgn_1^1 \prec_\infty bgn_2^1\big)\big\}\\
&= \big\{\big(u_1^1 \wedge u_2^2,\ \{bgn_2^2 \prec_\infty bgn_1^2,\ bgn_1^1 \prec_\infty bgn_2^1\}\big),\\
&\qquad \big(u_1^1 \wedge \neg u_2^2,\ bgn_2^2 \prec_\infty bgn_1^2\big),\ \big(u_2^2 \wedge \neg u_1^1,\ bgn_1^1 \prec_\infty bgn_2^1\big)\big\}\ .
\end{aligned}$$

According to property 2.4, both mutual exclusion and atomicity are ensured by the sum of both priority functions:

$$\begin{aligned}
pr_{mutex} \oplus pr_{atom} &= \big(u_1^{12} \wedge u_2^2,\ bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}\big),\ \big(u_2^{12} \wedge w_1,\ bgn_1^1 \prec_\infty end_2\}\big),\\
&\quad \big(u_1^{12} \wedge w_2,\ bgn_2^2 \prec_\infty end_1\}\big),\ \big(u_2^{12} \wedge u_1^1,\ bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}\big),\\
&\quad \big\{\big(u_1^1 \wedge u_2^2,\ \bot\big)\big\}\ \oplus\ pr_{atom}\\
&= \big\{\big(u_1^1 \wedge u_2^{12},\ bgn_2^2 \prec_\infty bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}\big),\\
&\quad \big(u_1^{12} \wedge u_2^2,\ bgn_1^1 \prec_\infty bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}\big),\\
&\quad \big(u_1^1 \wedge \neg(u_2^2 \vee u_2^{12}),\ bgn_2^2 \prec_\infty bgn_1^2\big),\\
&\quad \big(u_2^2 \wedge \neg(u_1^1 \vee u_1^{12}),\ bgn_1^1 \prec_\infty bgn_2^1\big),\\
&\quad \big(u_2^{12} \wedge w_1,\ bgn_1^1 \prec_\infty end_2\big),\ \big(u_1^{12} \wedge w_2,\ bgn_2^2 \prec_\infty end_1\big),\\
&\quad \big(u_1^1 \wedge u_2^2,\ \bot\big)\big\}\ .
\end{aligned}$$

The resulting priority function shows that both priority functions pr_{mutex} and pr_{atom} are compatible except from the state (u_1^1, u_2^2), where pr_{mutex} is not defined. Theorem 3.3 makes sure that this state is never reached. Therefore, the priority function $pr_{\mathsf{mutex}} \oplus pr_{\mathsf{atom}}$ ensures mutual exclusion on $\|_\emptyset\{TS_1, TS_2\}$ while preventing the system from reaching the deadlocking state (u_1^1, u_2^2).

3.2.6 Global Maximal Progress

In the product timed system $\|_\Sigma\big\{(TS_i, pr_i) \mid i \in \{1, \ldots, n\}\big\}$, synchronization actions have unbounded priority over the interleaving actions, provided that the synchronizing components are in a control state where the synchronization is possible. In contrast to this notion of *local* maximal progress, *global* maximal progress means that a synchronizing action a_i is prevented from interleaving as long as some other component TS_j synchronizing with a_i can progress, whatsoever the control state of TS_j.

For any component TS_i with action set A_i, let $A_i^- = \{a_i \in A_i \mid \exists \mathbf{a} \in \Sigma \,.\, a_i \in \mathbf{a}\}$ denote the set of synchronizing or *incomplete* actions, and $A_i^+ = (A_i \setminus A_i^-) \cup \{\mathbf{a} \in \Sigma \mid \mathbf{a} \cap A_i \neq \emptyset\}$ be the set of *complete* actions of TS_i. Global maximal progress is modelled by the priority function

$$pr_{\mathsf{maxprg}} = \bigoplus_{i \neq j} \big\{\big(true,\ \{a_i \prec_\infty a_j \mid \exists \mathbf{a} \in A_i^+ \cap A_j^+ \,.\, a_i \in A_i^- \cap \mathbf{a} \ \wedge\ a_j \in A_j^+\}\big)\big\}\ .$$

Intuitively, any incomplete action $a_i \in A_i^-$ synchronizing with an action of some component TS_j is disabled as long as TS_j can execute complete actions, so that TS_j can eventually synchronize with the waiting component TS_i.

3.2.7 Obtaining the Synchronized System

The timed system with priorities (TS, pr_{sync}) of synchronized processes is obtained by composing the processes and restricting the global system according to the steps described above. Formally,

$$\begin{aligned}(TS, pr_{\mathsf{sync}}) \quad = \quad & \big(\|_{\Sigma_{PMTN} \cup \Sigma}\big\{(TS_i^{PMTN},\ pr_i) \mid i \in \{1, \ldots, n\}\big\}, \\ & pr_{\mathsf{mutex}} \oplus pr_{\mathsf{atom}} \oplus pr_{\mathsf{maxprg}}\big)\ .\end{aligned}$$

According to property 2.4, the timed system (TS, pr_{sync}) satisfies all of the mutual exclusion, atomicity, and maximal progress properties modelled by pr_{mutex}, pr_{atom}, and pr_{maxprg}.

Notations: We characterize some action sets on TS, which will be needed for describing the scheduler. Let $AL_{i,r}$ denote the actions of TS allocating $r \in R$ to P_i: if r is not preemptable, $AL_{i,r} = BGN_{i,r}$; if r is preemptable, $AL_{i,r} = \big\{b_i \,|\, p_{i,r},\ b_i \,|\, p_{i,r} \,|\, pt_{j,r},\ rs_{i,r} \mid b_i \in BGN_{i,r} \wedge j \in \{1, \ldots, n\}\big\}$. Let $PMT_{i,j}^r = \{b_i \,|\, p_{i,r} \,|\, pt_{j,r} \mid b_i \in BGN_{i,r}\}$ denote the actions of P_i preempting P_j from $r \in R_p$.

Chapter 4

Liveness by Construction

Building systems which satisfy given specifications is a central problem in systems engineering. Standard engineering practice consists in decomposing the system to be designed into a set of cooperating components or processes. A key problem is the coordination of the components so that the global behaviour satisfies given specifications. Usually, ad hoc design methodologies are used leading to solutions that must be validated by verification, in particular deadlock detection [Cor96a], and testing. In some cases, it is possible to solve the coordination problem by synthesizing a controller or supervisor that restricts the behaviour of the components [AMP95, MPS95, AGP$^+$99, TA99, KLP$^+$98]. Both verification and synthesis techniques have well-known limitations due to their inherent complexity or undecidability, and cannot be applied to complex systems. As an alternative to cope with complexity, compositional description techniques have been studied.

As we have seen in section 2.5, it is necessary for a timed system modelling a real-time system, to be *timelock-free* that is, in any maximal run time diverges. Another essential property for timed systems modelling real-time applications such as controllers, schedulers, etc. is that any maximal run contains infinitely many actions. We call this property *livelock-freedom* as it implies deadlock-freedom and excludes indefinite waiting, and we call *live* a timed system which is both timelock-free and livelock-free.

We propose a method for building live systems as the composition of live components. The method is based on the key idea of enforcing the satisfaction of properties by appropriate structural restrictions preserved by composition operations. This leads to consider structural properties, intrinsic properties of the system which can be checked locally at low cost. We define a structural property called *structural liveness* which implies liveness and can be easily checked on components as the conjunction of three more elementary structural properties. We combine parallel composition and priority choice to build structurally live systems from structurally live components.

There exists some work about building deadlock-free reactive systems, such as [AAE98, CMSV99, IMA99]. [Bes94] discusses modelling real-time systems with time-constrained reactive automata whose expressiveness is restricted so that only physically sound systems can be specified. First steps towards the verification of structural properties of timed systems have been made in [Tri99], but no guarantees for the preservation of liveness are given. More recently, the concept of I/O timed components introduced in [Bra00] ensures liveness by proposing in each synchronization, a set of alternative actions one of which is always enabled. In all cases, guaranteeing liveness by structural properties requires relatively strong hypotheses.

The results below have been shown in [BGS00] for timed systems with deadlines as in section 2.4.1, for which all clock rates are equal to one. Here we generalize these results to timed systems, and complete them with some additional results.

4.1 Structural Liveness

We define *structural liveness* as a conjunction of structural properties ruling out physically unsound systems, and ensuring liveness. The properties are sufficiently "robust" to be preserved by parallel composition and priority choice, if certain precautions are taken.

Definition 4.1 (Left-closure) *An X-constraint C is* left-closed *with respect to a rate vector* $\mathbf{b}$ *if* $\forall \mathbf{x} \in \mathbb{D}^m$,

$$\big(\forall d \in \mathbb{D}_{>0}\ \exists d' \in \mathbb{D},\ d' < d \ .\ C(\mathbf{x} + d'\mathbf{b})\big) \implies C(\mathbf{x})\ .$$

Notice that for $\mathbb{D} = \mathbb{N}$, any X-constraint is left-closed.

Let $TS = (S, A, T, X, \mathbf{b}, h)$ be a timed system. We define the set of *initial states* of TS as a structural property.

Definition 4.2 (Initial clock valuations) *Let $s \in S$ be a control state of TS, and $\{t_i = (s_i, a_i, s)\}_{i \in I}$ with $h(t_i) = (g_i, \tau_i, r_i)$ be the set of transitions with destination s. If $I \neq \emptyset$, then the set of* initial clock valuations *for s is defined as the X-constraint*

$$in_s = \bigvee_{i \in I} post(a_i)$$

where $post(a_i)$ is the most liberal post-condition of the i-th transition defined by

$$post(a_i)(\mathbf{x}) = \exists \mathbf{x}' \in \mathbb{D}^m\ .\ g_i(\mathbf{x}') \wedge \mathbf{x} = \mathbf{x}'[r_i]\ .$$

When $I = \emptyset$, we take in_s to correspond to the valuation where all the clocks are set to zero.

A state $(s, \mathbf{x})$ such that $in_s(\mathbf{x})$ is called an initial state.

Notice that in most practical cases where guards are linear constraints, the quantifier in the expression of $post(b_i)$ can be eliminated.

Example 4.1 *For the timed system TS of figure 4.1 modelling a periodic process, $in_{sl} = (2 \leqslant x_p \leqslant 3\ \wedge\ t_p \leqslant 5)$, $in_w = (t_p = 0)$, and $in_c = (x_p = 0)$.*

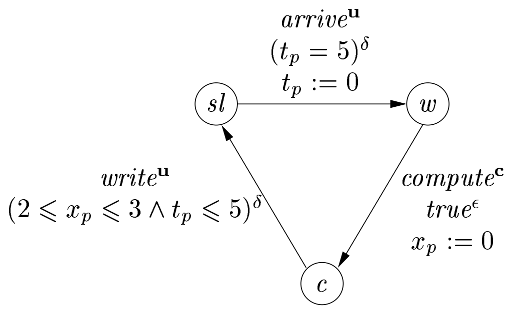


Figure 4.1: Timed system *TS* modelling a periodic process.

Property 4.1 (Time additivity) *If for some* $d_1, d_2 \in \mathbb{D}$ *and some state* $(s, \mathbf{x})$, $(s, \mathbf{x}) \stackrel{d_1+d_2}{\rightarrow} (s, \mathbf{x} + (d_1 + d_2)\mathbf{b}_s)$, *then* $(s, \mathbf{x}) \stackrel{d_1}{\rightarrow} (s, \mathbf{x} + d_1\mathbf{b}_s) \stackrel{d_2}{\rightarrow} (s, \mathbf{x} + (d_1 + d_2)\mathbf{b}_s)$, *and conversely.*

Proof. The property follows directly from the definition of time progress in timed systems in definition 2.3. ∎

Definition 4.3 (Run) *A* run *of a timed system is a maximal sequence of alternating time steps and discrete transitions starting from* $(s_0, \mathbf{x}_0)$ *such that* $in_{s_0}(\mathbf{x}_0)$*:*

$$(s_0, \mathbf{x}_0) \stackrel{d_0}{\rightarrow} (s_0, \mathbf{x}_0 + d_0\mathbf{b}_{s_0}) \stackrel{a_1}{\rightarrow} (s_1, \mathbf{x}_1) \stackrel{d_1}{\rightarrow} \ldots \stackrel{a_i}{\rightarrow} (s_i, \mathbf{x}_i) \stackrel{d_i}{\rightarrow} (s_i, \mathbf{x}_i + d_i\mathbf{b}_{s_i}) \stackrel{a_{i+1}}{\rightarrow} \ldots$$

Notice that due to time additivity of the timed system, any sequence of time steps can be reduced into a time step of cumulated duration. Therefore, any execution sequence of the model can be represented as a run (where some time steps may be of duration zero). If from some state $(s, \mathbf{x}) \in S \times \mathbb{D}^m$ indefinite waiting is allowed, we write $(s, \mathbf{x}) \stackrel{\infty}{\rightarrow} (s, \infty)$.

We now come to the definition of liveness of a timed system.

Definition 4.4 (Timelock-freedom) *A timed system is* timelock-free *if in any run* $(s_0, \mathbf{x}_0) \stackrel{d_0}{\rightarrow} (s_0, \mathbf{x}_0 + d_0\mathbf{b}_{s_0}) \stackrel{a_1}{\rightarrow} (s_1, \mathbf{x}_1) \stackrel{d_1}{\rightarrow} \ldots,\quad \sum_i d_i$ *diverges.*

Definition 4.5 (Livelock-freedom) *A timed system is* livelock-free *if in any run some action occurs infinitely often.*

Definition 4.6 (Liveness) *A timed system is called* live *if it is both timelock-free and livelock-free.*

We now introduce three structural properties whose conjunction implies liveness: structural non-zenoness, local timelock-freedom, and local livelock-freedom. Table 4.1 shows the relation between the different properties.

$$\left.\begin{array}{l}\left.\begin{array}{r}\textit{local timelock-freedom}\\ \textit{structural non-zenoness}\end{array}\right\} \Longrightarrow \textit{timelock-freedom}\\ \left.\begin{array}{r}\textit{local timelock-freedom}\\ \textit{local livelock-freedom}\end{array}\right\} \Longrightarrow \textit{livelock-freedom}\end{array}\right\} \Longrightarrow \textit{liveness}$$

Table 4.1: Three structural properties as a sufficient condition for liveness.

Definition 4.7 (Structural non-zenoness) *A timed system is* structurally non-zeno *if in any circuit of the discrete transition graph at least one clock is reset, and it is tested against some positive lower bound.*

Structural non-zenoness implies that there is a positive lower bound to the execution time of any circuit, and is the same as strong non-zenoness in [Tri99]. The timed system *TS* of figure 4.1 is structurally non-zeno because of the inter-arrival time of 5.

Definition 4.8 (Deadline) *The* deadline *of a transition* $t = (s, a, s') \in T$ *with* $h(t) = (g, \tau, r)$ *is the* X*-constraint characterizing the clock valuations* $\mathbf{x} \in \mathbb{D}^m$ *for which* t *blocks time progress according to the semantics of definition 2.3, that is,*

$$dl_t(\mathbf{x}) = \begin{cases} g(\mathbf{x}) \wedge \forall d \in \mathbb{D}_{>0} \,.\, \neg g(\mathbf{x} + d\mathbf{b}_s) & \textit{if} \quad \tau = \delta \\ \forall d \in \mathbb{D}_{>0}\ \exists d' \in \mathbb{D},\ d' < d \,.\, g(\mathbf{x} + d'\mathbf{b}_s) & \textit{if} \quad \tau = \epsilon \\ \textit{false} & \textit{if} \quad \tau = \lambda\,. \end{cases}$$

An X-constraint C is called *bounded* with respect to a rate vector $\mathbf{b}$ if $\forall \mathbf{x} \in \mathbb{D}^m\ \exists d \in \mathbb{D} \,.\, \neg(\Diamond^{\mathbf{b}} C)(\mathbf{x} + d\mathbf{b})$. For some control state $s \in S$, let $\textit{guards}_s = \bigvee_{t_i \in T(s)} g_i$, and $\textit{guards}_s^{\delta\epsilon} = \bigvee_{t_i \in T^{\delta\epsilon}(s)} g_i$, where $T^{\delta\epsilon}(s) = \{t_i \in T(s) \mid \tau_i = \epsilon \ \vee\ \tau_i = \delta \ \wedge\ g_i \textit{ is bounded w.r.t. } \mathbf{b}_s\}$, and $h(t_i) = (g_i, \tau_i, r_i)$.

Definition 4.9 (Local timelock-freedom) *A timed system is called* locally timelock-free *if for any state* $s \in S$ *and any transition* $t = (s, a, s') \in T(s)$ *issued from* s, $dl_t \implies \textit{guards}_s$.

This means that at any state $(s, \mathbf{x})$ of a locally timelock-free timed system, time can progress or some action is enabled.

Example 4.2 *Consider some waiting state* s *with two leaving transitions: transition* t_1 *acquiring some resource and guarded by* $(x \leqslant 5)^\delta$*, and a timeout transition* t_2 *guarded by* $(x > 5)^\epsilon$*. We have* $\textit{guards}_s = \textit{true}$*,* $dl_{t_1} = (x = 5)$*, and* $dl_{t_2} = (x \geqslant 5)$ *(for both continuous and discrete time). Therefore, the condition for timelock-freedom is verified in* s*. Without transition* t_1*, we would have* $\textit{guards}_s = (x > 5)$*, and the condition would not hold.*

Proposition 4.1 (Local timelock-freedom) *Timed systems where all eager transitions are left-closed are locally timelock-free.*

Proof. Suppose that all eager transitions of TS are left-closed, and let $(s, \mathbf{x}) \in S \times \mathbb{D}^m$ be some state of TS from which time cannot progress by any positive amount, that is, $dl_t(\mathbf{x})$ for some $t \in T(s)$. We show that t is enabled. If t is delayable with $h(t) = (g, \delta, r)$, then $dl_t(\mathbf{x})$ implies $g(\mathbf{x})$.

If t is eager with $h(t) = (g, \epsilon, r)$, then $\forall d \in \mathbb{D}_{>0}\ \exists d' \in \mathbb{D},\ d' < d\ .\ g(\mathbf{x} + d'\mathbf{b}_s)$. By hypothesis, g is left-closed, which means that $\big(\forall d \in \mathbb{D}_{>0}\ \exists d' \in \mathbb{D},\ d' < d\ .\ g(\mathbf{x} + d'\mathbf{b})\big) \implies g(\mathbf{x})$. The conjunction of both conditions implies $g(\mathbf{x})$. In both cases, $guards_s(\mathbf{x})$ holds. ■

Consequently, the timed system TS of figure 4.1 is timelock-free.

Lemma 4.1 (Timelock-freedom) *Any structurally non-zeno and locally timelock-free timed system is timelock-free.*

Proof. Local timelock-freedom guarantees that runs are either of infinite length or end with indefinite waiting. To prove timelock-freedom, we only have to check that in runs of infinite length, time diverges. This is a consequence of the fact that there is a finite number of control states, and that the system is structurally non-zeno. ■

Definition 4.10 (Local livelock-freedom) *A timed system is called* locally livelock-free *if for any state s, $in_s \implies \Diamond^{\mathbf{b}_s} guards_s^{\delta\epsilon}$. That is, from any initial clock valuation some transition will become urgent.*

Example 4.3 *Let us check state by state if the timed system TS of figure 4.1 is locally livelock-free.*

$$\begin{aligned}
in_{sl} = (2 \leqslant x_p \leqslant 3 \ \wedge\ t_p \leqslant 5) &\implies \Diamond^{\mathbf{b}_{sl}} guards_{sl}^{\delta\epsilon} = \Diamond^{\mathbf{b}_{sl}}(t_p = 5) = (t_p \leqslant 5) \\
in_w = (t_p = 0) &\implies \Diamond^{\mathbf{b}_w} guards_w^{\delta\epsilon} = true \\
in_c = (x_p = 0) &\;\not\!\!\implies \Diamond^{\mathbf{b}_c} guards_c^{\delta\epsilon} = \Diamond^{\mathbf{b}_c}(2 \leqslant x_p \leqslant 3 \ \wedge\ t_p \leqslant 5) \\
&\qquad = (x_p \leqslant 3 \ \wedge\ t_p \leqslant 5 \ \wedge\ t_p - x_p \leqslant 3)
\end{aligned}$$

Thus, TS is not locally timelock-free, since the guard transition compute entering state c does not contain enough information about the clock t_p measuring the time elapsed since the beginning of the period. Indeed, if execution in state c begins too late, it can not be completed before the end of the period. Therefore, we strengthen the guard of transition compute by $t_p \leqslant 5 - 3$, that is, the difference between the period and the maximal execution time in state c. We have then

$$in_c = (t_p \leqslant 2 \ \wedge\ x_p = 0)$$

which, this time, does imply $\Diamond^{\mathbf{b}_c} guards_c^{\delta\epsilon}$. It remains to verify that the condition in state w is still verified:

$$\Diamond^{\mathbf{b}_w} guards_w^{\delta\epsilon} = \Diamond^{\mathbf{b}_w}(t_p \leqslant 2) = (t_p \leqslant 2)$$

is still satisfied by $in_w = (t_p = 0)$. Therefore, the obtained timed system is locally livelock-free. It is easy to verify that it is still structurally non-zeno, and structurally timelock-free.

Proposition 4.2 (Livelock-freedom) *Any locally timelock-free and locally livelock-free timed system is livelock-free.*

Proof. Local livelock-freedom guarantees that from state $(s, \mathbf{x})$ such that $in_s(\mathbf{x})$, an action will eventually be executed if time can progress. Local timelock-freedom guarantees that time can progress until some deadline is reached. The proof follows from the fact that after execution of a transition the system reaches some state $(s', \mathbf{x}')$ such that $in_{s'}(\mathbf{x}')$. ■

Definition 4.11 (Structural liveness) *A timed system is* structurally live *if it is locally timelock-free, locally livelock-free, and structurally non-zeno.*

Clearly, structural liveness is a particular case of liveness that can be characterized by means of three properties easy to check. These three properties ensure common-sense requirements relying time progress and occurrence of events in the timed system: time must always diverge (local timelock-freedom); only a finite number of events can occur within some finite amount of time (structural non-zenoness); the system will always progress (local livelock-freedom). For instance, the modified timed system of example 4.3 is structurally live. We believe that structural liveness is a question of a "good modelling style" in the framework of timed systems.

Example 4.4 *Let TS_i be a structurally live timed system modelling a process according to section 3.1. Let us check if $(TS_i^{PMTN}, pr_i) = \left(\|_{\Sigma_i}\{TS_i\} \cup \{PMTN_{i,r} \mid r \in R_p \ \wedge\ U_{i,r} \neq \emptyset\}\right) \,/\, K_i,\ pr'_i)$ is structurally live. For some control state $s = (s_1, \dots)$ of TS_i^{PMTN}, where s_1 is the state of component TS_i, let $guards_{s_1}$ denote the union of guards of transitions issued from s_1 in TS_i.*

Structural non-zenoness: *TS_i is structurally non-zeno. For preemptable resources $r \in R_p$, the timed systems $PMTN_{i,r}$ have two loops that could potentially introduce Zeno behaviour in TS_i^{PMTN}: $\{p_{i,r}, v_{i,r}\}$, and $\{pt_{i,r}, rs_{i,r}\}$. The actions of the first loop synchronize with actions in $BGN_{i,r}$ and $END_{i,r}$, respectively, and do not interleave. According to the construction of TS_i, there is a clock $x_{i,r}$ measuring the execution time on r which is reset by actions in $BGN_{i,r}$, and tested against the positive lower bound $E^l_{i,r}$ at the guard of actions in $END_{i,r}$. Both conditions still hold when the actions in $BGN_{i,r}$ and $END_{i,r}$ are synchronized with the $p_{i,r}$ and $v_{i,r}$ actions, respectively.*

As to $(s, pt_{i,r}, s')$ and $(s, rs_{i,r}, s'')$ transitions, the guards of the former are restricted by pr'_i to $x_{i,r} \leqslant E^u_{i,r} \wedge \neg\Diamond guards_{s_1}$. Since there is no clock reset in this circuit, (TS_i^{PMTN}, pr_i) is not structurally non-zeno unless $x_{i,r} \leqslant E^u_{i,r} \wedge \neg\Diamond^{\mathbf{b}_{s_1}} guards_s = false$, that is, $x_{i,r} \leqslant E^u_{i,r} \implies \Diamond^{\mathbf{b}_s} guards_{s_1}$.

Local timelock-freedom: *We verify that the construction of (TS_i^{PMTN}, pr_i) does not introduce any left-open eager guards. TS_i is locally timelock-free, that is, for any timed transition $t = (s_1, a, (g, \tau, r), s'_1)$ of TS_i, $dl_t \implies guards_{s_1}$. This also holds for the non-synchronizing transitions $pt_{i,r}$ and $rs_{i,r}$ of $PMTN_{i,r}$. Transitions in $BGN_{i,r}$ and $END_{i,r}$ are synchronized with $p_{i,r}$ and $v_{i,r}$ transitions, respectively.*

The guards of $b_i \shortmid p_{i,r}$ transitions, where $b_i \in BGN_{i,r}$, are not restricted and keep the urgency type of the b_i transition. The guards of actions in $END_{i,r}$ synchronizing with $v_{i,r}$ actions are delayable, and therefore satisfy the local livelock-freedom constraint. Therefore, (TS_i^{PMTN}, pr_i) is locally timelock-free.

Local livelock-freedom: *TS_i is locally livelock-free. Consider some control state s of TS_i^{PMTN}. We check local livelock-freedom in s for each possible state of $PMTN_{i,r}$, for any resource $r \in R_p$. If $n\_use_{i,r}(s)$, then the transitions entering and leaving s have the same guards and urgency as in TS_i, such that (TS_i^{PMTN}, pr_i) is locally livelock-free in s. If $active_{i,r}(s)$, then there is, in addition to the transitions of TS_i, an entering transition $rs_{i,r}$ guarded by $x_{i,r} \leqslant E_{i,r}^u$, and a lazy leaving transition $pt_{i,r}$ guarded by $x_{i,r} \leqslant E_{i,r}^u \wedge \neg\Diamond guards_{s_1}$. It follows that (TS_i^{PMTN}, pr_i) is locally livelock-free in s if $x_{i,r} \leqslant E_{i,r}^u \implies \Diamond^{\mathbf{b}_{s_1}} guards_{s_1}^{\delta\epsilon}$. Finally, the choice of the guards, and the urgency of the leaving transition, guarantee locally livelock-freedom of (TS_i^{PMTN}, pr_i) in s if $susp_{i,r}(s)$.*

Therefore, (TS_i^{PMTN}, pr_i) is in general not structurally live. However, it follows from the discussion above that (TS_i^{PMTN}, pr_i) is structurally live if the guards of transitions $(s, pt_{i,r}, s')$ and $(s, rs_{i,r}, s'')$ are restricted to $\Diamond^{\mathbf{b}_{s_1}} guards_{s_1}^{\delta\epsilon}$, that is, the precondition for the bounded delayable or eager transitions leaving s_1 in TS_i to become enabled.

Hypothesis: In the sequel we require that priority functions pr satisfy the following conditions:

- For any action $a \in A$ that is not minimal in pr, all transitions labelled by a are either delayable and bounded, or eager. This requirement means that non-minimal actions eventually become urgent. It serves to preserve local livelock-freedom.
- For $(C, \prec) \in pr$, C is time-invariant, that is, for any control state s, and $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{D}^m$ such that $\mathbf{x}_2 = x_1 + d\mathbf{b}_s$ for some $d \in \mathbb{D}$, $C(s, \mathbf{x}_1) \iff C(s, \mathbf{x}_2)$. This condition is needed to preserve local timelock-freedom.

4.1.1 Preservation of Structural Liveness

We now come to the results allowing compositional reasoning about liveness of timed systems.

Theorem 4.1 (Liveness preservation under priority choice) *If pr is a well-defined priority function, and $TS = (S, A, T, X, \mathbf{b}, h)$ is a structurally live timed system where all eager guards are left-closed, then (TS, pr) is structurally live.*

Proof. **Structural non-zenoness:** priority choice with a well-defined priority function does not change the discrete transition structure of TS, and does not affect resets. So any circuit in (TS, pr) is a circuit in TS, and the clock resets are the same. Moreover, guards are restricted by priority orders, so a lower bound in a guard may only increase. Consequently, if TS is structurally non-zeno, then (TS, pr) is structurally non-zeno, too.

Local timelock-freedom: consider some control state $s \in S$. Priority choice transforms a left-closed guard g_1 either into a left-closed guard g_1', or there is according to definition 2.8 some higher-priority action with guard g_2 such that $g_1' \vee g_2$ is left-closed. By induction on the priority order it can be shown that $\bigvee_{t_i \in T(s)} g_i'$ is left-closed. Therefore, $dl_t \implies guards_s$ for any $t \in T(s)$, that is, (TS, pr) is locally timelock-free.

Local livelock-freedom: priority choice with a well-defined priority function does not change the discrete transition structure of TS, and restricts only guards. Consequently, for each state $s \in S$, the set of transitions entering s does not change, and in_s is restricted. If we note in_s' the set of initial clock valuations of s in (TS, pr), and as TS is locally livelock-free, we have $in_s' \implies in_s \implies \Diamond^{\mathbf{b}_s} guards_s^{\delta\epsilon}$.

If the guard g_1 of some delayable or eager transition t_1 is restricted to g_1' by some transition t_2, then the latter is delayable and bounded, or eager, and we have $\Diamond^{\mathbf{b}_s} g_1 \implies \Diamond^{\mathbf{b}_s}(g_1' \vee g_2)$. By induction on the priority order it follows that $\Diamond^{\mathbf{b}_s} guards_s^{\delta\epsilon} \implies \Diamond^{\mathbf{b}_s}(guards_s^{\delta\epsilon})'$, where the primed expression refers to (TS, pr). Thus, $in_s \implies \Diamond^{\mathbf{b}_s}(guards_s^{\delta\epsilon})'$, that is, local livelock-freedom is preserved. ■

Theorem 4.2 (Liveness preservation under parallel composition) *Let (TS_i, pr_i), $i \in \{1, \ldots, n\}$, be timed systems with priorities, and Σ be a set of synchronization actions as in definition 2.13. If the timed systems $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$ are structurally live and have all guards left-closed, and the associated priority functions pr_i are well-defined, then the timed system $(TS, pr) = \|_\Sigma \{(TS_i,\ pr_i) \mid i \in \{1, \ldots, n\}\}$ is structurally live and has its guards left-closed.*

Proof. The priority functions pr_i are well-defined and defined on disjoint action sets A_i, and interleaving actions are minimal in pr_Σ. Therefore, the priority function $pr = pr_\Sigma \oplus \bigoplus_{i \in \{1,\ldots,n\}} pr_i$ is well-defined. By application of theorem 4.1, it is sufficient to show that TS is structurally live and has its guards left-closed.

Structural non-zenoness: since each transition (interleaving or synchronization) corresponds to a transition in one or more components, each circuit in TS contains a set of transitions forming a circuit in some TS_i. If all components TS_i are structurally non-zeno, then in all these circuits some clock is reset and tested against a positive lower bound. Then this is the case for all the circuits of TS too. The bounds of a synchronization action may increase, but cannot decrease.

Local timelock-freedom: conjunction and disjunction preserve left-closure of guards, so guards of synchronization transitions are left-closed as well as those of interleaving transitions. This guarantees local timelock-freedom.

Local livelock-freedom: Interleaving transitions are transitions of the components, and synchronization guards are conjunctions of guards of the components. No delayable or eager action is dominated by a lazy action. If all TS_i are locally livelock-free, we have for $s = (s_1, \ldots, s_n) \in S$,

$$in_s \implies \bigvee_{i \in \{1,\ldots,n\}} in_{s_i} \implies \bigvee_{i \in \{1,\ldots,n\}} \Diamond guards_{s_i}^{\delta\epsilon} \implies \Diamond guards_s^{\delta\epsilon} .$$

Thus, the product system is locally livelock-free. ■

The urgency type *eager* is a convenient way of specifying that some transition is to be taken "as soon as possible". Under the aspect of structural liveness, eager transitions with bounded guards can be replaced by delayable transitions without losing structural liveness.

Lemma 4.2 (Liveness preservation under elimination of eager transitions) *If TS is a structurally live timed system, then the system $TS^{\lambda\delta}$, that is identical with TS except from the fact that all eager transitions with bounded guards of TS are delayable in $TS^{\lambda\delta}$, is structurally live.*

Proof. **Structural non-zenoness** is clearly preserved.

Local timelock-freedom: for delayable transitions $t = (s, a, s')$ of $TS^{\lambda\delta}$ with $h(t) = (g, \delta, r)$, $dl_t \implies g \implies guards_s$. Since TS is locally timelock-free, $TS^{\lambda\delta}$ is locally timelock-free, too.

Local livelock-freedom: is preserved since for any $s \in S$, the X-constraint $guards_s^{\delta\epsilon}$ does not change. ■

Definition 4.12 (Projection) *Let $(TS, pr) = \|_\Sigma\{TS_i, pr_i \mid i \in \{1, \ldots, n\}\}$ be the parallel composition of the timed systems $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$, $i \in \{1, \ldots, n\}$, with synchronization actions Σ. The* projection *of the transition relation $\rightarrow^{TS}$ on the component TS_i, $i \in \{1, \ldots, n\}$, is the transition relation $\pi_i(\rightarrow^{TS})$ defined as follows. For any clock valuation $\mathbf{x}$ of TS, let $\mathbf{x}|_i$ denote the restriction of $\mathbf{x}$ to the clocks in X_i.*

$$\begin{aligned}
\pi_i(\rightarrow^{TS}) \quad = \quad & \big\{(s_i, \mathbf{x}|_i) \xrightarrow{d} (s_i, (\mathbf{x} + d\mathbf{b}_s)|_i) \mid (s, \mathbf{x}) \xrightarrow{d}{}^{TS} (s, \mathbf{x} + d\mathbf{b}_s) \wedge d \in \mathbb{D}\big\} \cup \\
& \big\{(s_i, \mathbf{x}|_i) \xrightarrow{a} (s_i', \mathbf{x}'|_i) \mid (s, \mathbf{x}) \xrightarrow{a}{}^{TS} (s', \mathbf{x}') \wedge a \in A_i\big\} \cup \\
& \big\{(s_i, \mathbf{x}|_i) \xrightarrow{a} (s_i', \mathbf{x}'|_i) \mid (s, \mathbf{x}) \xrightarrow{\mathbf{a}}{}^{TS} (s', \mathbf{x}') \wedge \mathbf{a} \in \Sigma \wedge a \in \mathbf{a} \cap A_i\big\}
\end{aligned}$$

Intuitively, time steps, and discrete transitions of TS_i in TS are preserved by π_i. Synchronization transitions are projected on the actions of TS_i. Transitions in which TS_i is not involved, are abstracted away by π.

The following proposition shows that liveness preservation under parallel composition is not obtained at the price of adding new transitions. In other words, if some action can be executed in the parallel composition of timed systems, then it can be executed in at least one of the components, modulo some waiting.

Proposition 4.3 (Behaviour inclusion) *Let $(TS, pr) = \|_\Sigma\{(TS_i, pr_i) \mid i \in \{1, \ldots, n\}\}$ be the parallel composition of the timed systems $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$*

with priorities for $i \in \{1, \ldots, n\}$, with pairwise disjoint clock sets, and synchronization actions Σ such that only transitions with bounded guards synchronize. Then,

$$\pi_i(\rightarrow^{TS}) \subseteq \rightarrow^{(TS_i, pr_i)^{\lambda\delta}}$$

for any $i \in \{1, \ldots, n\}$.

Proof. For discrete transitions, the inclusion follows from definition 2.13 of the parallel composition. Therefore, we focus on time steps. Let $(s, \mathbf{x}) \xrightarrow{d}{}^{TS} (s, \mathbf{x} + d\mathbf{b}_s)$.

According to the semantics of TS (definition 2.3), this implies that (1) for any delayable transition issued from s with guard g, $\neg(\lozenge^{\mathbf{b}_s} g)(\mathbf{x}) \;\vee\; (\lozenge^{\mathbf{b}_s} g)(\mathbf{x} + d\mathbf{b}_s)$, and that (2) for any eager transition issued from s with guard g, $\forall d' \in \mathbb{D},\ d' < d \,.\, \neg g(\mathbf{x} + d'\mathbf{b}_s)$. We have to show that $(s_i, \mathbf{x}|_i) \xrightarrow{d}{}^{(TS_i, pr_i)^{\lambda\delta}} \big(s_i, (\mathbf{x} + d\mathbf{b}_s)|_i\big)$. Notice that $(\mathbf{x} + d\mathbf{b}_s)|_i = \mathbf{x}|_i + d\mathbf{b}_i(s_i)$, because the clock sets are disjoint, such that according to definition 2.13, only zero values in $\mathbf{b}_i$ can stop the clocks of TS_i in TS.

- For any eager transition with guard g in $(TS_i, pr_i)^{\lambda\delta}$, we have $\forall d' \in \mathbb{D},\ d' < d \,.\, \neg g\big((\mathbf{x} + d'\mathbf{b}_s)|_i\big)$, since the only eager transitions in $(TS_i, pr_i)^{\lambda\delta}$ do not synchronize, and are interleaving transitions in TS.
- Similarly, for any delayable transition of $(TS_i, pr_i)^{\lambda\delta}$ that does not synchronize in s, we have $\neg(\lozenge^{\mathbf{b}_s} g)(\mathbf{x}|_i) \vee (\lozenge^{\mathbf{b}_s} g)\big((\mathbf{x} + d\mathbf{b}_s)|_i\big)$. This also holds if the transition was eager in (TS_i, pr_i).
- For any delayable transition of $(TS_i, pr_i)^{\lambda\delta}$ with guard g_i such that $\neg(\lozenge^{\mathbf{b}_s} g_i)(\mathbf{x}|_i) \vee (\lozenge^{\mathbf{b}_s} g_i)\big((\mathbf{x} + d\mathbf{b}_s)|_i\big)$ that synchronizes in s, let g be the guard of the synchronization transition (thus, $g \implies g_i$), and $g_i' = g_i \wedge \neg\lozenge^{\mathbf{b}_s} g$ be the guard of the interleaving transition in (TS, pr). According to the hypothesis, we have (1) $\neg(\lozenge^{\mathbf{b}_s} g)(\mathbf{x}) \;\vee\; (\lozenge^{\mathbf{b}_s} g)(\mathbf{x} + d\mathbf{b}_s)$ — this is also the case if the transition is eager in (TS, pr) —, and (2) $\neg(\lozenge^{\mathbf{b}_s} g_i')(\mathbf{x}) \;\vee\; (\lozenge^{\mathbf{b}_s} g_i')(\mathbf{x} + d\mathbf{b}_s)$ in (TS, pr). Suppose that $(\lozenge^{\mathbf{b}_s} g_i)(\mathbf{x}|_i) \;\wedge\; \neg(\lozenge^{\mathbf{b}_s} g_i)\big((\mathbf{x} + d\mathbf{b}_s)|_i\big)$. This implies that $\big((\lozenge^{\mathbf{b}_s} g)(\mathbf{x}) \;\vee\; (\lozenge^{\mathbf{b}_s} g_1')(\mathbf{x})\big) \;\wedge\; \neg(\lozenge^{\mathbf{b}_s} g)(\mathbf{x} + d\mathbf{b}_s) \;\wedge\; \neg(\lozenge^{\mathbf{b}_s} g_i')(\mathbf{x} + d\mathbf{b}_s)$, which is in contradiction to (1) and (2).

Therefore, $(s_i, \mathbf{x}|_i) \xrightarrow{d}{}^{(TS_i, pr_i)^{\lambda\delta}} \big(s_i, (\mathbf{x} + d\mathbf{b}_s)|_i\big)$. ■

Thus, components preserve their behaviour when composed with other timed systems, modulo some waiting introduced by the maximal progress paradigm. This waiting is bounded, as shown in the next section.

4.2 Individual Liveness

An important property in the previous example is individual livelock-freedom of the components in the product system. Liveness of the product does not imply that in any run, an action of some component occurs infinitely often. This remark motivates the following definition and theorem.

Definition 4.13 (Individual livelock-freedom) *A component* (TS_j, pr_j) *is* live-lock-free *in a timed system* $(TS, pr) = \|_\Sigma \{(TS_i, pr_i) \mid i \in \{1, \ldots, n\}\}$ *if in each run of* (TS, pr) *with initial state* $((s_1, \ldots, s_n), \mathbf{x})$ *such that* $(s_i, \mathbf{x}|_i)$ *is an initial state of* TS_i *for all* $i \in \{1, \ldots, n\}$*, some action of* (TS_j, pr_j) *occurs infinitely often.* $\mathbf{x}|_i$ *denotes the restriction of* $\mathbf{x}$ *to the clocks in* X_i.

Theorem 4.3 (Individual livelock-freedom under parallel composition) *Let* (TS_i, pr_i), $i \in \{1, \ldots, n\}$*, be structurally live timed systems with priorities having disjoint clock sets, and let* Σ *be a set of synchronization actions as in definition 2.13. If the guard of any synchronizing transition* (s_i, a, s_i') *of any component* TS_i *is bounded with respect to* $\mathbf{b}_i(s_i)$*, then all components* TS_i *are livelock-free in* $(TS, pr) = \|_\Sigma \{(TS_i, pr_i) \mid i \in \{1, \ldots, n\}\}$.

Proof. See appendix A.

Guaranteeing the preservation of individual livelock-freedom under priority choice requires some additional assumption on the component timed systems which makes sure that priority choice does not disable all actions of some component to the profit of other components.

Definition 4.14 (Persistent local livelock-freedom) *A control state* s *is called* persistently locally livelock-free *with respect to a priority function* pr *if* $in_s \implies \Diamond^{\mathbf{b}_s} guards_s^{\delta\epsilon,\max}$*, where* $guards_s^{\delta\epsilon,\max}$ *is* $guards_s^{\delta\epsilon}$ *where only transitions labelled by actions that are maximal in* pr*, and whose synchronization actions are maximal in* pr*, are counted.*

This property means that at state s, maybe after some waiting, it is always possible to execute an action. It is instrumental for avoiding deadlocks in components when guards are restricted by using priorities.

Theorem 4.4 (Individual livelock-freedom under priority choice) *Let* (TS_i, pr_i), $i \in \{1, \ldots, n\}$*, be structurally live timed systems with priorities having disjoint clock sets, let* Σ *be a set of synchronization actions as in definition 2.13, and let* pr' *be a priority function. If some component* TS_i *is persistently locally livelock-free with respect to* pr'*,* TS_i *is livelock-free in* $(TS, pr) = \|_\Sigma \{(TS_i, pr_i) \mid i \in \{1, \ldots, n\}\}$*, and* $pr \oplus pr'$ *is well-defined, then* TS_i *is livelock-free in* $(TS, pr \oplus pr')$.

Proof. The proof is similar to the proof of theorem 4.3. Instead of choosing some transition $t = (s_1, a, s_1')$ of (TS_1, pr_1) with $h_1(t) = (g, \tau, r)$, $(\Diamond^{\mathbf{b}_{s_1}} g)(\mathbf{x}_1)$, and either $\tau = \delta$ and g is bounded, or $\tau = \epsilon$, take among the transitions satisfying the above condition, some transition t that is labelled by an action a that is maximal in pr'. Such a transition exists because of the persistent local livelock-freedom of TS_i with respect to pr'. ■

Example 4.5 (Mutual exclusion) *Let* (TS_i, pr_i), $i \in \{1, \ldots, n\}$*, be structurally live timed systems with priorities and disjoint clock sets, and let* $U_{i,r}$ *and* $W_{i,r}$ *characterize* *their using and waiting states with respect to a resource $r \in R_n$. Moreover, let pr_{mutex} be the priority function ensuring mutual exclusion on R, as in section 3.2.4. If pr_{mutex} is well-defined, and the waiting states of some component TS_i are persistently locally livelock-free with respect to pr_{mutex}, then TS_i is livelock-free in $\big(\|_\emptyset\big\{(TS_i, pr_i) \mid i \in \{1,\ldots,n\}\big\},\ pr_{\mathsf{mutex}}\big)$.*

The persistence of the waiting states makes sure that no process becomes deadlocked while waiting to enter the using states. The finite execution times are needed to satisfy the hypothesis of bounded guards of synchronizing actions.

Theorem 4.5 (Individual livelock-freedom under hierarchical composition) *Let $(TS_i, pr_i) = \|_{\Sigma_i}\big\{(TS_{ij}, pr_{ij}) \mid j \in \{1,\ldots,n_i\}\big\}$ be the parallel composition of structurally live timed systems with priorities having disjoint clock sets, and let Σ be a set of synchronization actions as in definition 2.13. If the guard of any synchronizing transition (s_{ij}, a, s'_{ij}) of any component TS_{ij} is bounded with respect to $\mathbf{b}_{ij}(s_{ij})$, then any component TS_{ij} is livelock-free in $(TS, pr) = \|_\Sigma\big\{(TS_i, pr_i) \mid i \in \{1,\ldots,n\}\big\}$.*

Moreover, if some component TS_{ij} is persistently locally livelock-free with respect to a priority function pr', and $pr \oplus pr'$ is well-defined, then TS_{ij} is livelock-free in $(TS, pr \oplus pr')$.

Proof. According to remark 2.1,

$$\begin{aligned}
(TS, pr) &= \|_\Sigma\big\{(TS_i, pr_i) \mid i \in \{1,\ldots,n\}\big\} \\
&= \|_\Sigma\Big\{\|_{\Sigma_i}\big\{(TS_{ij}, pr_{ij}) \mid j \in \{1,\ldots,n_i\}\big\} \mid i \in \{1,\ldots,n\}\Big\} \\
&= \|_{\Sigma \cup \bigcup_{i\in\{1,\ldots,n\}} \Sigma_i}\big\{(TS_{ij}, pr_{ij}) \mid i \in \{1,\ldots,n\} \wedge j \in \{1,\ldots,n_i\}\big\}\ .
\end{aligned}$$

Livelock-freedom of all components in (TS, pr) follows from theorem 4.3. Livelock-freedom of any persistently locally livelock-free component in $(TS, pr \oplus pr')$ follows from theorem 4.4. ∎

4.3 Structural Safety

In contrast to formalisms where safety properties are ensured by construction — for example, by using semaphores for mutual exclusion, or action constraints [AGS00] —, modelling safety properties using priority functions relies on the fact that transitions violating the property, are disabled by dominating actions that become eventually enabled. Hence, safety is closely related to liveness in our model.

The following definition provides a sufficient condition for a timed system with priority to be safe with respect to the invariance of a set of control states. The condition is structural, and therefore inexpensive to check.

Definition 4.15 (Structural safety) *Let $(TS, pr) = \|_\Sigma\big\{(TS_i, pr_i) \mid i \in \{1,\ldots,n\}\big\}$, where $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$, $i \in \{1,\ldots,n\}$, are timed systems with disjoint clock sets, and Σ is a set of synchronization actions as in definition 2.13. Furthermore, if $TS = (S, A, T, X, \mathbf{b}, h)$, let S' be a subset of the control states S of TS such that*

$\forall(s, \mathbf{x}) \in S' \times \mathbb{D}^m \cdot pr(s, \mathbf{x}) \neq \bot$. *Let* $unsafe_{S'} = \{(s, a, s') \in T \mid s \in S' \wedge s' \notin S'\}$ *denote the set of transitions leaving* S', *and let*

$$\begin{aligned} safe_{S'} \;=\; & \{t = (s, a, s') \in T \mid s, s' \in S' \;\wedge\; \exists i \in \{1, \ldots, n\} \,.\, a \in A_i \\ & \wedge\, h(t) = (g, \tau, r) \;\wedge\; (\tau = \epsilon \,\vee\, \tau = \delta \wedge g \textit{ is bounded}) \;\wedge\; in_{s_i} \implies \Diamond^{\mathbf{b}_{s_i}} g \\ & \wedge\, \forall(s'', \mathbf{x}) \in S' \times \mathbb{D}^m \,.\, a \textit{ is maximal in } pr(s'', \mathbf{x})\} \,. \end{aligned}$$

(TS, pr) *is* structurally safe *with respect to* S' *if for any* $(s, a_1, s_1) \in unsafe_{S'}$, *there is some transition* $(s, a_2, s_2) \in safe_{S'} \cap T(s)$ *with* $\forall \mathbf{x} \in \mathbb{D}^m \,.\, \{a_1 \prec_\infty a_2\} \subseteq pr(s, \mathbf{x})$.

If (TS, pr) is structurally safe with respect to S', then any transition leaving S' is disabled by pr giving unbounded priority to one or more transitions remaining in S'. $safe_{S'}$ denotes the set of transitions issued from some state $s \in S'$ and remaining in S' (first line) that eventually become enabled and urgent in the component TS_i (second line) — and also in (TS, pr), since the clock sets are disjoint —, and their guards are not restricted by pr (third line). Figure 4.2 illustrates the notion of structural safety.

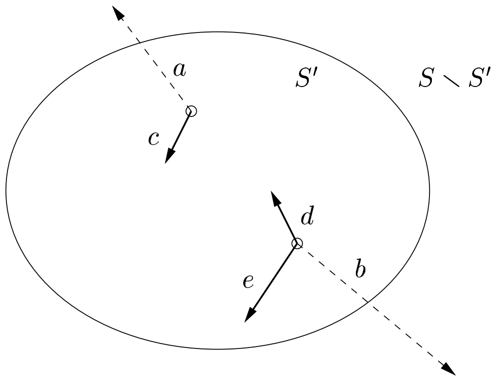


Figure 4.2: Structural safety: any of the transitions a and b leaving S' is dominated by at least one transition issued from the same state, and remaining in S'.

According to the property of activity preservation, priority choice preserves deadlock-freedom in control states for which the priority function is well-defined. Structural safety is an inexpensive test for the invariance of these states, and thus for the preservation of deadlock-freedom.

Example 4.6 *In example 3.6, the mutual exclusion constraints on the two resources create a deadlock in state* (u_1^1, u_2^2) *of the timed system. Let us examine if the synchronized system* $\left(\|_\emptyset \{TS_1, TS_2\}, pr_{\mathsf{mutex}}\right)$ *with the timing constraints shown in figure 4.3, is structurally safe with respect to the states where mutual exclusion is verified. With*

$$\begin{aligned} pr_{\mathsf{mutex}} \;=\; & \{(u_2^{12} \wedge u_1^1,\; bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}),\; (u_1^{12} \wedge u_2^2,\; bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}), \\ & (u_2^{12} \wedge w_1,\; bgn_1^1 \prec_\infty end_2\}),\; (u_1^{12} \wedge w_2,\; bgn_2^2 \prec_\infty end_1\}), \\ & (u_1^1 \wedge u_2^2,\; \bot)\} \,, \end{aligned}$$

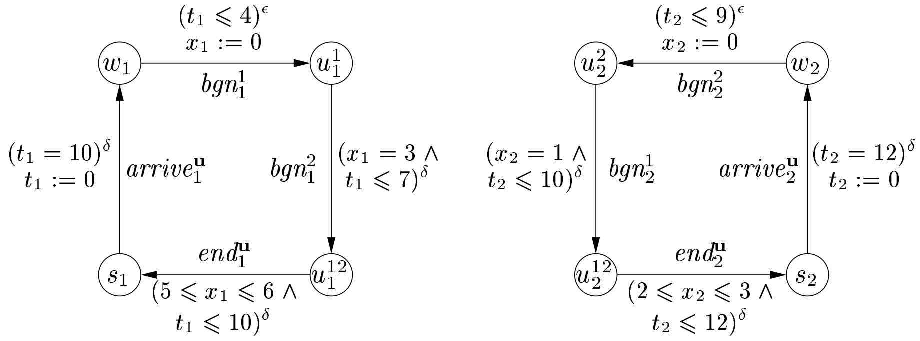


Figure 4.3: Two processes sharing two non-preemptable resources.

the safe states where moreover pr_{mutex} *is well-defined are*

$$S' = \{s_1, w_1, u_1^1, u_1^{12}\} \times \{s_2, w_2, u_2^2, u_2^{12}\} \setminus \{(u_1^1, u_2^2), (u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12})\} \ .$$

According to definition 4.15 we obtain

$$\begin{aligned} unsafe_{S'} &= \big\{\big((w_1, u_2^2), bgn_1^1, (u_1^1, u_2^2)\big),\ \big((u_1^1, w_2), bgn_2^2, (u_1^1, u_2^2)\big)\big\} \\ safe_{S'} \cap T\big((w_1, u_2^2)\big) &= \big\{\big((w_1, u_2^2), bgn_2^1, (w_1, u_2^{12})\big)\big\} \\ safe_{S'} \cap T\big((u_1^1, w_2)\big) &= \big\{\big((u_1^1, w_2), bgn_1^2, (u_1^{12}, w_2)\big)\big\} \ . \end{aligned}$$

Since the transitions in $unsafe_{S'}$ *are not dominated by transitions in* $safe_{S'}$ *according to* pr_{mutex}*, the synchronized system* $\big(\|_\emptyset\{\,TS_1, TS_2\,\}, pr_{\mathsf{mutex}}\big)$ *is not structurally safe with respect to* S'*.*

Theorem 4.6 (Safety) *If* (TS, pr) *is structurally safe with respect to some set of control states* S' *according to definition 4.15, and* $(s, \mathbf{x}) \subseteq S' \times \mathbb{D}^m$ *is a state of* TS *such that* $\big(\forall t = (s, a, s') \in safe_{S'} \cap T(s)$*, where* $h(t) = (g, \tau, r)$*,* $(\Diamond^{\mathbf{b}_s} g)(\mathbf{x})\big)$*, then any run of* (TS, pr) *with initial state* $(s, \mathbf{x})$ *remains within* S'*.*

Proof. We first show that for any state $(s', \mathbf{x}')$ reachable by a run starting from $(s, \mathbf{x})$,

$$\forall t = (s', a, s'') \in safe_{S'} \cap T(s') \text{ with } h(t) = (g, \tau, r) \ . \ (\Diamond^{\mathbf{b}_{s'}} g)(\mathbf{x}') \tag{4.1}$$

holds. By hypothesis, (4.1) holds for $(s, \mathbf{x})$. Let $(s', \mathbf{x}')$ be some state reached from $(s, \mathbf{x})$ by a timed or discrete transition. If $(s, \mathbf{x}) \xrightarrow{a} (s', \mathbf{x}')$ with $a \in A$ and $s' = (s'_1, \ldots, s'_n)$, then $in_{s'_i}(\mathbf{x}'|_i)$ for any component that participates in a. The clocks of any other component are not affected. Thus, (4.1) holds for $(s', \mathbf{x}')$. If $(s, \mathbf{x}) \xrightarrow{d} (s, \mathbf{x}')$ with $d \in \mathbb{D}$, then none of the transitions $t = (s, a, s') \in safe_{S'} \cap T(s)$ has become urgent yet. Since $(\Diamond^{\mathbf{b}_s} g)(\mathbf{x})$, a is maximal in pr, and t eventually becomes urgent, (4.1) holds for $(s, \mathbf{x}')$.

From any state $(s, \mathbf{x})$ satisfying (4.1), some transition $t \in safe_{S'} \cap T(s)$ will eventually become enabled and urgent, such that pr disables the transitions in $unsafe_{S'} \cap T(s)$. Therefore, no transition can lead TS from S' to $S \setminus S'$. ∎

Example 4.7 *In example 3.6, the mutual exclusion constraints on the two resources create a deadlock in state (u_1^1, u_2^2) of the timed system. We have solved this problem in example 3.8 by declaring the allocation of both resources by one process atomic, and obtained the priority function*

$$\begin{aligned} pr_{\mathsf{mutex}} \oplus pr_{\mathsf{atom}} \;=\; & \big\{ \big(u_1^1 \wedge u_2^{12},\ bgn_2^2 \prec_\infty bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}\big), \\ & \big(u_1^{12} \wedge u_2^2,\ bgn_1^1 \prec_\infty bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}\big), \\ & \big(u_1^1 \wedge \neg(u_2^2 \vee u_2^{12}),\ bgn_2^2 \prec_\infty bgn_1^2\big), \\ & \big(u_2^2 \wedge \neg(u_1^1 \vee u_1^{12}),\ bgn_1^1 \prec_\infty bgn_2^1\big), \\ & \big(u_2^{12} \wedge w_1,\ bgn_1^1 \prec_\infty end_2\big),\ \big(u_1^{12} \wedge w_2,\ bgn_2^2 \prec_\infty end_1\big), \\ & \big(u_1^1 \wedge u_2^2,\ \bot\big)\big\} \ . \end{aligned}$$

Let us now check safety of the synchronized system $\big(\|_\emptyset\{TS_1, TS_2\}, pr_{\mathsf{mutex}} \oplus pr_{\mathsf{atom}}\big)$ as shown in figure 4.3, with respect to the non-reachability of states (u_1^1, u_2^{12}), (u_1^{12}, u_2^2), and (u_1^{12}, u_2^{12}) where mutual exclusion is violated, and the state (u_1^1, u_2^2) where the system is deadlocked. Therefore, the safe states are

$$S' = \{s_1, w_1, u_1^1, u_1^{12}\} \times \{s_2, w_2, u_2^2, u_2^{12}\} \setminus \{(u_1^1, u_2^2), (u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12})\} \ ,$$

and we obtain

$$\begin{aligned} unsafe_{S'} &= \big\{\big((w_1, u_2^2), bgn_1^1, (u_1^1, u_2^2)\big),\ \big((u_1^1, w_2), bgn_2^2, (u_1^1, u_2^2)\big)\big\} \\ safe_{S'} \cap T\big((w_1, u_2^2)\big) &= \big\{\big((w_1, u_2^2), bgn_2^1, (w_1, u_2^{12})\big)\big\} \\ safe_{S'} \cap T\big((u_1^1, w_2)\big) &= \big\{\big((u_1^1, w_2), bgn_1^2, (u_1^{12}, w_2)\big)\big\} \end{aligned}$$

as before. This time, any transition in $unsafe_{S'}$ is dominated by some transition in $safe_{S'}$ according to $pr_{\mathsf{mutex}} \oplus pr_{\mathsf{atom}}$, that is, $\big(\|_\emptyset\{TS_1, TS_2\}, pr_{\mathsf{mutex}} \oplus pr_{\mathsf{atom}}\big)$ is structurally safe with respect to S'. According to theorem 4.6, the synchronized system remains in S' for any state that is the product of initial states of both processes, and whose control state is in S', such as for example $\big((s_1, u_2^2), (t_1, x_1, t_2, x_2) = (10, 5, 8, 0)\big)$.

4.3.1 A Priority Synthesis Algorithm

In the previous chapter, standard constructions for preemption, mutual exclusion, and atomicity, all based on priority functions, have been presented. Although the constructions are rather intuitive, one can ask how to find them, or how to resolve a different problem for which no solution has been proposed, while remaining in the framework of timed systems with priorities.

Another practical question is: having obtained a system that fails to be structurally safe, how to make it structurally safe? This is for example the case for a partially modelled system where only some safety properties are ensured. Or else, having applied several priority functions to a timed system and detected inconsistencies potentially causing the system to deadlock, how can the states with a circuit in the associated priority relation be made unreachable by *adding* a priority function to the existing one? This second point is closely related to the problem of deadlock avoidance [Gar00, Bel90].

Both problems are addressed by the following algorithm generalizing the constructions for mutual exclusion and atomicity presented in chapter 3. For a given timed system with priorities (TS, pr), say, $TS = (S, A, T, X, \mathbf{b}, h)$, and a set of *bad* states $B \subseteq S$ to be made unreachable, it computes a priority function pr' and a maximal set $S' \subseteq S \setminus B$ of control states such that

- (TS, pr') is structurally safe with respect to S';
- pr' "contains" pr, that is, $\forall (s, \mathbf{x}) \in S \times \mathbb{R}^m_{\geqslant 0} \,.\, pr(s, \mathbf{x}) \subseteq pr'(s, \mathbf{x}) \vee pr'(s, \mathbf{x}) = \perp$.

TS is supposed to be the product of timed systems with priorities, that is, $(TS, \tilde{pr}) = \|_\Sigma \{ (TS_i, pr_i) \mid i \in \{1, \ldots, n\} \}$, where $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$, $i \in \{1, \ldots, n\}$, are timed systems with disjoint clock sets, and Σ is a set of synchronization actions as in definition 2.13. pr may, but needs not be equal to $\tilde{pr}$, and does not need to be well-defined.

Algorithm 4.1 Priority synthesis for structural safety.

Input: a timed system $TS = (S, A, T, X, \mathbf{b}, h)$ product of timed systems $TS_i = (S_i, A_i, T_i, X_i, \mathbf{b}_i, h_i)$, $i \in \{1, \ldots, n\}$, a priority function pr, and a set $B \subseteq S$ of states to be made unreachable.
Output: a set $S' \subseteq S \setminus B$ of control states, and a priority function pr' such that (TS, pr') is structurally safe with respect to S'.

$pr' := pr$;
$S' := S \setminus \big(B \,\cup\, \{s \in S \mid \exists \mathbf{x} \in \mathbb{R}^m_{\geqslant 0} \,.\, pr'(s, \mathbf{x}) = \perp\}$;
$todo := \{s \in S' \mid \exists (s, a, s') \in T \,.\, s' \notin S'\}$;
for all $s \in todo$ **do**
    for all $t = (s, a_1, s_1) \in unsafe_s$ **do**
        let $safe := \big\{(s, a_2, s_2) \in safe_s \mid \forall \mathbf{x} \in \mathbb{D}^m \,.\, pr'(s, \mathbf{x}) \oplus \{a_1 \prec_\infty a_2\} \neq \perp\big\}$;
        if (a is controllable **and** $safe \neq \emptyset$)
            $pr' := pr' \oplus \big\{\big(s,\ a_1 \prec_\infty actions\ (safe)\big)\big\}$;
        else
            $S' := S' \setminus \{s\}$;
            $todo := todo \cup \{s' \in S' \mid \exists (s', a, s) \in T\}$;
        fi
    od
    $todo := todo \setminus \{s\}$;
od
return (pr', S').

How it works. S' is initialized as the set of "legal" states in $S \setminus B$ for which pr is well-defined. For a control state $s \in S'$, $safe_s$ and $unsafe_s$ denote the sets of transitions $safe_{S'} \cap T(s)$ and $unsafe_{S'} \cap T(s)$ issued from s, respectively. We call s safe if each transition issued from s that leaves S', is disabled by some transition in $safe_s$. $todo$ is the set of control states whose safety remains to be established, and is initialized with the states in S' having a successor in $S \setminus S'$. The safety of the states in $S' \setminus todo$ has been verified. As long as there exists some state $s \in todo$ such that there exists some transition $t = (s, a_1, s_1) \in unsafe_s$, the set $safe \subseteq safe_s$ is computed, containing the transitions that can be given priority over t while keeping pr' well-defined in s. If $safe$ is not empty, and a_1 is controllable, then the actions in $safe$ are given unbounded priority over a_1. Otherwise, the state s cannot be guaranteed to be safe and is taken out of S', and safety of its predecessor states must be (re-)verified. When the algorithm terminates with $todo = \emptyset$, (TS, pr') is structurally safe with respect to S'.

If all components (TS_i, pr_i) are structurally live and have all guards left-closed, and pr' is well-defined, then theorem 4.1 guarantees (TS, pr') to be structurally live.

Notice that the algorithm does not require the timed system TS to be effectively constructed; all operation can be carried out on the components and the priority function.

Optimality. The algorithm only assigns priorities disabling transitions leaving S', and excludes only states from S' that are unsafe. Therefore, it is optimal with respect to structural safety, in the sense that it returns the maximal set S' such that (TS, pr') is structurally safe with respect to S'.

The algorithm is not optimal with respect to safety, though, in the sense that it may not find a set S' and a priority function pr' such that (TS, pr') is safe with respect to S', although such a tuple (pr', S') exists. This is due to the fact that structural safety is a sufficient, but not necessary, condition for safety.

Complexity. In every round of the outer **for** loop, either a control state is deleted from S' and $O(|S'|)$ states are added to $todo$, or a control state is erased from $todo$. Let tps be an upper bound on the number of transitions issued from one control state of S. This means that any $s \in S$ has at most tps successors, and can be added to $todo$ at most tps times. Therefore, the outer **for** loop is executed at most $O(|S| \times tps)$ times.

For every execution of the body of the outer **for**-loop, both $unsafe_s$ and $safe_s$ contain at most $O(tps)$ transitions. If we make abstraction from the implementation-dependent cost of updating the priority function, the complexity of the algorithm is $O(|S| \times tps^3)$ in the worst case.

Example 4.8 *In example 3.6, the mutual exclusion constraints on the use two resources create a deadlock in state (u_1^1, u_2^2) of the timed system $TS = \|_{\emptyset}\{TS_1, TS_2\}$. We have solved this problem in example 3.8 by declaring the allocation of both resources by one process atomic.*

We will now use algorithm 4.1 to find a tuple (pr', S') such that (TS, pr') is structurally safe with respect to S', where we specify the set of bad states to be avoided as $B = \{(u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12})\}$, that is, the states in which mutual exclusion is violated. Figure 4.3 takes up the components of TS. Let us see step by step how the algorithm proceeds when called with TS as in example 3.6, $pr = \emptyset$, and B.

1. $pr' = \emptyset$
   $S' = S \setminus \{(u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12})\}$
   $todo = \{(w_1, u_2^{12}), (u_1^{12}, w_2), (u_1^1, u_2^2)\}$; *take* $s = (w_1, u_2^{12})$:
   $unsafe_s = \big\{\big((w_1, u_2^{12}), bgn_1^1, (u_1^1, u_2^{12})\big)\big\}, \quad safe_s = \big\{\big((w_1, u_2^{12}), end_2, (w_1, s_2)\big)\big\}.$

2. $pr' = \big\{\big((w_1, u_2^{12}), bgn_1^1 \prec_\infty end_2\big)\big\}$
   $S' = S \setminus \{(u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12})\}$
   $todo = \{(u_1^{12}, w_2), (u_1^1, u_2^2)\}$; *take* $s = (u_1^{12}, w_2)$:
   $unsafe_s = \big\{\big((u_1^{12}, w_2), bgn_2^2, (u_1^{12}, u_2^2)\big)\big\}, \quad safe_s = \big\{\big((u_1^{12}, w_2), end_1, (s_1, w_2)\big)\big\}.$

3. $pr' = \big\{\big((w_1, u_2^{12}), bgn_1^1 \prec_\infty end_2\big),\ \big((u_1^{12}, w_2), bgn_2^2 \prec_\infty end_1\big)\big\}$
   $S' = S \setminus \{(u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12})\}$
   $todo = \{(u_1^1, u_2^2)\}$; *take* $s = (u_1^1, u_2^2)$:
   $unsafe_s = \big\{\big((u_1^1, u_2^2), bgn_1^2, (u_1^{12}, u_2^2)\big), \big((u_1^1, u_2^2), bgn_2^1, (u_1^1, u_2^{12})\big)\big\}, \quad safe_s = \emptyset.$

4. $pr' = \big\{\big((w_1, u_2^{12}), bgn_1^1 \prec_\infty end_2\big),\ \big((u_1^{12}, w_2), bgn_2^2 \prec_\infty end_1\big)\big\}$
   $S' = S \setminus \{(u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12}), (u_1^1, u_2^2)\}$
   $todo = \{(w_1, u_2^2), (u_1^1, w_2)\}$; *take* $s = (w_1, u_2^2)$:
   $unsafe_s = \big\{\big((w_1, u_2^2), bgn_1^1, (u_1^1, u_2^2)\big)\big\}, \quad safe_s = \big\{\big((w_1, u_2^2), bgn_2^1, (w_1, u_2^{12})\big)\big\}.$

5. $pr' = \big\{\big((w_1, u_2^{12}), bgn_1^1 \prec_\infty end_2\big),\ \big((u_1^{12}, w_2), bgn_2^2 \prec_\infty end_1\big),\ \big((w_1, u_2^2), bgn_1^1 \prec_\infty bgn_2^1\big)\big\}$
   $S' = S \setminus \{(u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12}), (u_1^1, u_2^2)\}$
   $todo = \{(u_1^1, w_2)\}$; *take* $s = (u_1^1, w_2)$:
   $unsafe_s = \big\{\big((u_1^1, w_2), bgn_2^2, (u_1^1, u_2^2)\big)\big\}, \quad safe_s = \big\{\big((u_1^1, w_2), bgn_1^2, (u_1^{12}, w_2)\big)\big\}.$

6. $pr' = \big\{\big((w_1, u_2^{12}), bgn_1^1 \prec_\infty end_2\big),\ \big((u_1^{12}, w_2), bgn_2^2 \prec_\infty end_1\big),\ \big((w_1, u_2^2), bgn_1^1 \prec_\infty bgn_2^1\big), \big((u_1^1, w_2), bgn_2^2 \prec_\infty bgn_1^2\big)\big\}$
   $S' = S \setminus \{(u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12}), (u_1^1, u_2^2)\}$
   $todo = \emptyset$.

Intuitively, in steps 1 and 2 the algorithm adds to pr' *the priorities ensuring mutual exclusion on* r_1 *and* r_2*, respectively. In the third step, the blocking state* (u_1^1, u_2^2) *is found to be unsafe, and deleted from* S'*. In steps 4 and 5, this state is made unreachable, which amounts to making the successive allocation of both resources atomic. The timed system with priorities* (TS, pr') *is structurally safe with respect to* S'*. Since the guards of* TS_1 *and* TS_2 *are left-closed, and both timed systems are structurally live, theorem 4.1 guarantees* (TS, pr') *to be structurally live.*

Chapter 5

Modelling Real-Time Schedulers

Having constructed the synchronized system consisting of the set of processes, and their functional interaction constraints such as mutual exclusion and preemption, we shall now turn to the upper layer of the global system model, the scheduler. The role of the scheduler is to coordinate the access of processes to resources so as to ensure fairness, and guarantee the timing requirements to be met. Adopting the same philosophy of representing process interaction constraints as priority functions as in the previous chapters, we show how scheduling policies can be modelled as priority functions.

In real-time systems design, especially when dealing with critical or embedded systems, *predictability* of the scheduled system is an issue of growing importance. Protocols such as the priority ceiling protocol [SRL90] are designed to provide this predictability, but the complex priority inheritance mechanism makes verification a hard task. Formalizing scheduling policies as priority functions allows to check for consistency with other constraints. If correctness must be proven, the scheduled system modelled as a timed system with priorities can be model-checked, taking into account the scheduling policy applied to the system, and reducing its non-determinism.

5.1 Methodology

We suppose a timed system with priorities (TS, pr_{sync}) to be given that is composed of the timed systems modelling the processes $\{P_i \mid i \in \{1, \ldots, n\}\}$, as in chapter 3. We further suppose that R_n and R_p denote the set of non-preemptable and preemptable resources, respectively, and that (TS, pr_{sync}) respects mutual exclusion on all resources.

Many existing scheduling policies distinguish between rules assigning process priorities to resolve conflicts between processes on the one hand, and admission control rules deciding whether some process is eligible for resource allocation on the other hand [Kes97]. For example, the priority ceiling protocol schedules the process with the highest current priority among the processes that are waiting for the processor, whereas a process P_i is eligible for the allocation of a free resource if the current priority of P_i is higher than the priority ceilings of all resources currently allocated to processes other than P_i.

We formalize this decomposition of a scheduling policy as a conjunction of two priority functions: $pr_{\mathsf{pol}} = pr_{\mathsf{adm}} \oplus pr_{\mathsf{res}}$, where pr_{adm} defines an admission control that restricts resource allocations according to a given condition, whereas pr_{res} resolves conflicts between two or more processes waiting for the same resource.

In principle, for each resource, different admission control and conflict resolution policies are applicable. We take $pr_{\mathsf{adm}} = \bigoplus_{r \in R} pr^r_{\mathsf{adm}}$ and $pr_{\mathsf{res}} = \bigoplus_{r \in R} pr^r_{\mathsf{res}}$. For example, resources that are crucial for the correct functioning of a system could be allocated according to a conservative policy guaranteeing deadlock-freedom, whereas resources shared by less critical processes could be managed using a more optimistic policy in order to use them more efficiently. In practice, care should be taken to apply compatible policies so as to avoid inconsistency. Using the framework of priority functions to express the admission and conflict resolution policies for different resources, the composability results of priority functions guarantee composability of the different policies, and diagnostics in the case of their inconsistency.

5.2 Admission Control

The constraint $pr_{\mathsf{adm}} = \bigoplus_{r \in R} pr^r_{\mathsf{adm}}$ dynamically restricts the actions allocating r, depending on the current state of the application. We express it as a priority function, for the sake of homogeneity of the modelling formalism, which allows the designer to take advantage of the properties of priority functions. This means that in order to disable some transition, some "anchor" actions have to be identified that can be given priority over the actions that are to be disabled. In practice, the choice of such anchor actions is very natural in most cases.

Generally, the admission control policy specifies that some process P_j is not allowed to acquire some resource r as long as some condition on the resources held by other processes is satisfied, that is, as long as there exists some process P_i satisfying a predicate $blocking^r_{ij}$. In other words, there is some process P_i that *blocks* P_j on r as long as P_i has not freed one or more of the resources it holds, until the condition $blocking^r_{ij}$ is not verified any more. In this case, the admission control policy is implemented by a priority function giving priority to the actions of P_i using the resources it holds, or resuming its execution if it is preempted, over the actions allocating r to P_j, if $blocking^r_{ij}$ holds. Formally,

$$\begin{aligned} pr^r_{\mathsf{adm}} \quad = \quad & \bigoplus_{\substack{i,j \in \{1,\ldots,n\} \\ r' \in R}} \left\{ \left(blocking^r_{ij} \wedge U_{i,r'},\ AL_{j,r} \prec_\infty U_{i,r'} \circ \right) \right\} \oplus \\ & \bigoplus_{\substack{i,j \in \{1,\ldots,n\} \\ r' \in R_p}} \left\{ \left(blocking^r_{ij} \wedge susp_{i,r'},\ AL_{j,r} \prec_\infty rs_{i,r'} \right) \right\} . \end{aligned}$$

The examples in this chapter illustrate this idea.

Notice that in scheduling literature, the term *blocking* means particularly the blocking of a high-priority process by a low-priority process. Since the present framework is not restricted to fixed-priority scheduling, we understand by *blocking* the fact that according to the specification of the admission control policy to be modelled, some process prevents another process from acquiring a resource.

Example 5.1 *Coming back to example 3.6 of two processes sharing two non-preemptable resources, the potential deadlock in state* (u_1^1, u_2^2) *has been made unreachable in example 3.8 and example 4.8 by disallowing allocation of one resource to a process as soon as the other process has already acquired the other resource. This is a case of admission control: if there is a process* P_i *that holds resource* r_i*, then the other process is prevented from acquiring the other resource. This leads to the predicates*

$$\begin{array}{lcl} blocking_{12}^{r_1} & = & false \\ blocking_{12}^{r_2} & = & U_{1,r_1} \wedge W_{2,r_2} \;=\; (u_1^1 \vee u_1^{12}) \wedge w_2 \\ blocking_{21}^{r_1} & = & U_{2,r_2} \wedge W_{1,r_1} \;=\; (u_2^2 \vee u_2^{12}) \wedge w_1 \\ blocking_{21}^{r_2} & = & false \end{array}$$

and thus to the priority functions

$$\begin{array}{lcl} pr_{\mathsf{adm}}^{r_1} & = & \big\{\big((u_2^2 \vee u_2^{12}) \wedge w_1,\ bgn_1^1 \prec_\infty \{bgn_2^1, end_2\}\big)\big\} \\ pr_{\mathsf{adm}}^{r_2} & = & \big\{\big((u_1^1 \vee u_1^{12}) \wedge w_2,\ bgn_2^2 \prec_\infty \{bgn_1^2, end_1\}\big)\big\}\ . \end{array}$$

Their composition yields the priority function

$$\begin{array}{lcl} pr_{\mathsf{adm}} \;=\; pr_{\mathsf{adm}}^{r_1} \oplus pr_{\mathsf{adm}}^{r_2} & = & \big\{\big((u_2^2 \vee u_2^{12}) \wedge w_1,\ bgn_1^1 \prec_\infty \{bgn_2^1, end_2\}\big), \\ & & \big((u_1^1 \vee u_1^{12}) \wedge w_2,\ bgn_2^2 \prec_\infty \{bgn_1^2, end_1\}\big)\big\}\ . \end{array}$$

The effect of pr_{adm} *is identical with that of the priority function* $pr_{mutex} \oplus pr_{atom}$ *of example 3.8, since both priority functions differ only in some priorities between actions that are not possible under the associated conditions, and in the fact that* $pr_{mutex} \oplus pr_{atom}$ *is undefined in* (u_1^1, u_2^2)*. Therefore,* pr_{adm} *makes the allocation of both resources to one process atomic with respect to the allocation of the free resource to the other process. As a side effect,* pr_{adm} *expresses also mutual exclusion on the use of the resources, even if this is already ensured by the synchronization layer according to chapter 3.*

5.3 Conflict Resolution

We assume that for each resource r, conflicts are resolved according to a partial order on the set of processes $\{P_i \mid i \in \{1, \ldots, n\}\}$. The partial order is specified as a set of state constraints $\{\mathcal{C}_{ij}^r\}_{i,j \in \{1,\ldots,n\}}$. When $\mathcal{C}_{ij}^r$ holds, process P_i has priority over process P_j for using resource r. Notice that $\mathcal{C}_{ij}^r$ may depend on clock values as well as on control states. Three cases of conflict may appear.

First, if r is free, and two processes P_i and P_j are waiting for r, actions of $AL_{i,r}$ are given priority over actions of $AL_{j,r}$ by the priority rule

$$(pr_{\mathsf{res}}^r)' \;=\; \bigoplus_{i,j \in \{1,\ldots,n\}} \big\{\big(\mathcal{C}_{ij}^r,\ BGN_{j,r} \prec_0 BGN_{i,r}\big)\big\}\ .$$

Second, if r is preemptable, and there is a conflict between a process P_i using the resource, and a process P_j attempting to preempt it, preemption must be prevented if P_j is of lower priority. Preemption by lower-priority processes is prevented by restricting with the constraint

$$(pr^r_{\mathsf{res}})'' = \bigoplus_{i,j\in\{1,\ldots,n\}} \left\{ \left(\neg\mathcal{C}^r_{ji} \wedge active_{i,r},\ PMT^r_{j,i} \prec_\infty U_{i,r} \circ \right) \right\} .$$

Intuitively, if process P_i is active on r, actions of P_j preempting P_i from r are disabled by the actions making P_i progress unless P_j has priority over P_i.

Third, if r is preemptable, and a third process P_k is active on r that may be preempted by any of the processes P_i and P_j, then actions of P_i preempting P_k from r are given priority over actions of P_j preempting P_k from r:

$$(pr^r_{\mathsf{res}})''' = \bigoplus_{i,j,k\in\{1,\ldots,n\}} \left\{ \left(\mathcal{C}^r_{ij} \wedge \mathcal{C}^r_{ik} \wedge \mathcal{C}^r_{jk} \wedge active_{k,r},\ PMT^r_{j,k} \prec_0 PMT^r_{i,k}\right) \right\} .$$

The constraint pr^r_{res} specifying arbitration between processes according to the given order is obtained as a composition $pr^r_{\mathsf{res}} = (pr^r_{\mathsf{res}})' \oplus (pr^r_{\mathsf{res}})'' \oplus (pr^r_{\mathsf{res}})'''$, if $r \in R_p$, and $pr^r_{\mathsf{res}} = (pr^r_{\mathsf{res}})'$ if $r \in R_n$.

The synchronized system (TS, pr_{sync}), scheduled under the admission control and conflict resolution policies modelled by the priority functions pr^r_{adm} and pr^r_{res} for the resources $r \in R$, is of the form

$$\left(TS,\ pr_{\mathsf{sync}} \oplus \bigoplus_{r\in R} (pr^r_{\mathsf{adm}} \ \oplus\ pr^r_{\mathsf{res}})\right) .$$

The next sections give examples of how existing and widely used scheduling policies are modelled as priority functions in this general framework.

5.4 Specifying Well-Known Policies

We define some admission control and conflict resolution policies on the timed system (TS, pr_{sync}) composed of the timed systems modelling the processes $\{P_i \mid i \in \{1,\ldots,n\}\}$, as in chapter 3. For states where the policy does not specify a total order, we complete the policy order by the static order: P_i has priority over P_j if $i < j$.

5.4.1 Conflict Resolution Policies

The FIFO policy

The first-in-first-out policy (FIFO) on resource $r \in R$ specifies that r is granted to the process that has been waiting for the longest time. The FIFO policy is defined by

$$\mathcal{C}^r_{ij} \ = \ (t_i > t_j) \vee (t_i = t_j \wedge i < j) .$$

The first term of $\mathcal{C}^r_{ij}$ means that whenever two processes P_i and P_j are both waiting for r, P_i is served prior to P_j if process P_i has been waiting for longer time than process P_j, that is, $t_i > t_j$. The second term ensures a strict allocation order in case of conflict (that is, when $t_i = t_j$).

The EDF policy

The earliest-deadline-first policy (EDF) on resource $r \in R$ specifies that r is granted to the waiting process that is closest to its relative deadline [LL73]. The EDF policy is defined by

$$\mathcal{C}^r_{ij} \;=\; (D_{i,r} - t_i < D_{j,r} - t_j) \vee (D_{i,r} - t_i = D_{j,r} - t_j \wedge i < j) \;,$$

that is, whenever there are two processes P_i and P_j waiting for r, the actions granting r to P_i have immediate priority over the actions granting r to P_j if P_i is closer to its relative deadline than P_j (namely, $D_{i,r} - t_i < D_{j,r} - t_j$).

The RMS policy

The algorithm of preemptive rate-monotonic scheduling (RMS, [LL73]) assigns to each strictly periodic process a fixed priority such that processes with shorter period have higher priority, that is, if $T_i < T_j$, then P_i has priority over P_j. The RMS policy is specified by replacing $D_{i,r} - t_i < D_{j,r} - t_j$ in the EDF policy by the condition $T_i < T_j$.

5.4.2 The Priority Ceiling Protocol

When scheduling real-time systems, it is often crucial that an important, high-priority process be served within some known delay, even if it requests resources shared with other processes. Therefore, it is necessary to reduce undesired interference between processes, such as priority inversion, that can lead to system behaviours that are hard to predict [Ree97, CDKM00]. Priority inversion can take place if there are at least three processes, two of which share a non-preemptable resource r.

Example 5.2 (Priority inversion) *In example 3.7, three processes share a preemptable processor (CPU), and one non-preemptable resource r. Suppose that we want to schedule the three processes such that P_1 has the highest priority, and P_3 the lowest priority. In a first approach, the priority between the three processes could be modelled as follows: $BGN_{3,CPU} \prec_0 BGN_{2,CPU} \prec_0 BGN_{1,CPU}$ for transitions allocating the CPU, and $BGN_{3,r} \prec_0 BGN_{1,r}$ for resource allocations, which means that a higher-priority process has priority over a lower-priority process whenever both processes are willing to acquire the CPU or the resource, respectively. However, this model is prone to priority inversion. Suppose that P_3 is preempted by P_1 while using the resource. By executing a p_1 action to request the resource, P_1 frees the CPU and gets blocked on the resource. Now P_2 can acquire the CPU by a go_2 action, which leads the system into the state $s = (wr_1, e_2, u'_3)$ where priority inversion takes place: P_2 is running, instead of the high-priority process P_1 being served as soon as possible. P_1 has to wait for P_2 to complete its execution before P_3 can, in turn, complete and free the resource. In other words, P_2 has* de facto *priority over P_1.*

Several protocols have been proposed to avoid priority inversion [SRL90, RSLR95], the best-known of which is the priority ceiling protocol (PCP). It has been designed in order to improve the worst-case behaviour in the context of preemptive fixed-priority scheduling on a monoprocessor system with shared, non-preemptable resources. However, it requires a run-time scheduler performing relatively complex operations such as priority inheritance and priority updates. Therefore, formal verification is hard to carry out on a system of processes running under the ceiling protocol.

Although being rather restrictive [XP00], the ceiling protocol is widely used in real-time systems since it ensures some properties that help to make the timing of the system predictable. First, it avoids transitive blocking, that is, system states in which a process is waiting for a resource that is held by another process that cannot complete its execution because it is waiting for another resource allocated to a third process. Second, when a process is waiting for a resource, it is blocked for at most the time of one critical section. Finally, the ceiling protocol prevents cycles in the waiting graph, that is, deadlocks.

Little work deals with formalizing the priority ceiling protocol including timing issues. The PVS model in [Dut00] gives a clear formalization of the priority ceiling protocol in order to verify its properties. However, it does not aim at the verification of a given scheduled system of processes running under the priority ceiling protocol, and it adopts a simplified task model.

In this section, we model the priority ceiling protocol as a priority function. As in [SRL90], we consider a system composed of processes sharing a processor CPU, and a set of non preemptable resources R_n. The processor is the only preemptable resource, that is, $R_p = \{CPU\}$. We suppose that the process indices correspond to the process priorities such that a process of lower index has a higher priority.

In order to prevent priority inversion, priorities between actions allocating the CPU or resources have to be differentiated according to the global system state (in this example, according to whether P_1 is waiting for r or not).

Informally speaking, the priority ceiling protocol specifies that

(PCP-1) the current priority of a process is the maximum of its own fixed priority, and the priorities of the processes it blocks;

(PCP-2) a process can only obtain a resource if its current priority is higher than the ceiling of any resource allocated to some other process; and

(PCP-3) the process with the highest current priority is given the processor.

For a more formal specification of the priority ceiling protocol, see [SRL90].

For $r \in R_n$, let $c_r = \min\{i \mid i \in \{1, \ldots, n\} \;\wedge\; \exists s \in S_i \,.\, U_{i,r}(s)\}$ be the priority ceiling of r, that is, the index of the highest-priority process that may use r. Moreover, we suppose that the timed systems modelling the processes $P_1, \ldots, P_n$ satisfy the following conditions:

- No process requests a resource unless it holds the CPU: $\forall i \;\forall r \in R_n \,.\, \bullet W_{i,r} \subseteq U_{i,CPU}{}^{\circ}$.
- Waiting states do not use the CPU: $\forall i \;\forall r \in R_n \,.\, W_{i,r} \cap U_{CPU} = \emptyset$. As soon as a process requests a resource, its execution is suspended until the process acquires the resource. This rule allows a process holding the resource to complete its execution, and free the requested resource.

- A waiting state $s \in W_{i,r}$ can only be left by an action allocating r to P_i, modelling the fact that P_i is suspended until it obtains r.
- When acquiring a resource, a process must get the *CPU* at the same time: $\forall i\ \forall r \in R_n\ .\ BGN_{i,r} \subseteq BGN_{i,CPU}$.

It is easy to see that these conditions are verified in example 5.2.

Admission control

According to (PCP-2), a process P_j can acquire a resource only if its current priority is higher than the ceiling of any resource allocated to any other process. Conversely, some process P_i holding a resource r_1 of ceiling c_{r_1} prevents P_j from acquiring a resource (or *blocks* P_j) if the current priority of P_j is not strictly higher than c_{r_1}. Therefore, we can define a predicate $\widetilde{blocking}_{ij}$:

$$\widetilde{blocking}_{ij} \iff i \neq j\ \wedge\ \exists r_1, r_2 \in R_n\ .\ U_{i,r_1}\ \wedge\ W_{j,r_2}\ \wedge\ c_{r_1} \leqslant current(j)\ ,$$

where $current(j)$ denotes the current priority of P_j. The admission control policy of the priority ceiling protocol does not differentiate according to the resource P_j wants to acquire, that is, $blocking^r_{ij} = blocking_{ij}$ for all $r \in R_n$. According to (PCP-1), the current priority of P_j is the minimum of j, and the indices of the processes it blocks:

$$current(j)\ =\ \min\left\{j,\ \min\{k \mid \widetilde{blocking}_{jk}\}\right\}\ .$$

This description of an admission control policy requires a priority inheritance mechanism to keep the predicate *current* up to date. In order to define a predicate *blocking* in terms of a state constraint — that is, depending only on the current control states of the timed systems — we need to find a solution for the recursively defined predicate $\widetilde{blocking}$.

Before formulating a state predicate formalizing (PCP-2) and showing its equivalence with $\widetilde{blocking}$, we establish an inductive invariant under the admission control policy defined by $\widetilde{blocking}$ that simplifies the proofs of several properties of the priority ceiling protocol: if a process P_i holds a resource r_1 of priority ceiling c_{r_1}, then no process whose priority is "between" that of P_i and c_{r_1}, holds any resource. Formally,

$$\forall i, j\ \forall r_1, r_2 \in R_n\ .\ U_{i,r_1}\ \wedge\ c_{r_1} \leqslant j < i \implies \neg U_{j,r_2}\ . \tag{5.1}$$

Proposition 5.1 *Equation (5.1) is an inductive invariant under the admission control policy allowing the allocation of $r \in R_n$ to some process P_i if and only if there is no process P_j such that $\widetilde{blocking}_{ij}$.*

Proof. See appendix A.

The invariant (5.1) allows us to replace the predicate $current(j)$ in the definition of $\widetilde{blocking}_{ij}$ by the fixed priority j of P_j, without changing the behaviour of the admission control policy:

Proposition 5.2 (Admission control policy of the PCP) *The admission control policy of the priority ceiling protocol is modelled by the state constraint*

$$blocking_{ij} \iff i \neq j \ \wedge \ \exists r_1, r_2 \in R_n \ . \ U_{i,r_1} \ \wedge \ W_{j,r_2} \ \wedge \ c_{r_1} \leqslant j \ .$$

Proof. See appendix A.

Since a process that is *blocking* another process, holds a non-preemptable resource, the admission control policy (PCP-2) is modelled for any $r \in R_n$ by the priority function

$$\begin{aligned} pr^r_{\mathsf{adm}} \quad = \quad & \bigoplus_{\substack{i,j \in \{1,\ldots,n\} \\ r' \in R_n}} \left\{ \left(blocking_{ij} \wedge U_{i,r'}, \ AL_{j,r} \prec_\infty U_{i,r'} \circ \right) \right\} \ \oplus \\ & \bigoplus_{i,j \in \{1,\ldots,n\}} \left\{ \left(blocking_{ij} \wedge susp_{i,CPU}, \ BGN_{j,r} \prec_\infty rs_{i,CPU} \right) \right\} \ . \end{aligned}$$

If for any two processes P_i and P_j, $blocking_{ij}$ holds, then pr^r_{adm} gives priority to the actions of P_i using the resources it holds, or resuming its execution on the CPU if it has been preempted, over the actions allocating r to P_j (remember that for $r \in R_n$, $AL_{i,r} = BGN_{i,r}$).

Access to the CPU is not subject to an admission control policy; it is arbitered by the conflict resolution policy discussed below.

Conflict resolution

According to (PCP-3), the process with the highest current priority is given the processor. Formally,

$$\widetilde{\mathcal{C}^{CPU}_{ij}} \iff current(i) < current(j) \ .$$

This definition of a conflict resolution policy requires a priority inheritance mechanism in order to keep the predicate *current* up to date. However, we can show that under the admission control policy pr_{adm} defined above, the conflict resolution policy can be expressed using only predicates on the current control states of the processes.

Proposition 5.3 (Conflict resolution of the PCP)

$$\begin{aligned} \mathcal{C}^{CPU}_{ij} \quad \iff \quad & i < j \ \wedge \ \forall k \leqslant i \ . \ \neg blocking_{jk} \ \vee \\ & j < i \ \wedge \ \exists k < j \ . \ blocking_{ik} \end{aligned}$$

models the conflict resolution policy of the priority ceiling protocol.

Proof. See appendix A.

Intuitively, a process P_i is given priority over another process P_j if either P_i is of higher fixed priority than P_j, and P_j does not block any process of equal or higher priority than P_i; or the fixed priority of P_i is lower than that of P_j, but P_i blocks some process of priority higher than that of P_j.

Notice that $\mathcal{C}_{ij}^{CPU}$ is a state constraint that does not require a priority inheritance mechanism.

Access to the resources in R_n is constrained by the admission control policy pr_{adm} defined above; there is no conflict resolution policy for the resources.

Example 5.3 (Prevention of priority inversion under the PCP) *Consider the three processes P_1, P_2, P_3 of example 3.7. The ceiling priority of the resource r shared between P_1 and P_3 is $\min\{1,3\} = 1$. The predicates defined above become*

$$\begin{array}{rclrcl}
\mathit{blocking}_{12} & = & \mathit{false} & \mathit{blocking}_{21} & = & \mathit{false} \\
\mathit{blocking}_{13} & = & (u_1 \vee u_1') \wedge wr_3 & \mathit{blocking}_{31} & = & (u_3 \vee u_3') \wedge wr_1 \\
\mathit{blocking}_{23} & = & \mathit{false} & \mathit{blocking}_{32} & = & \mathit{false} \\
\mathcal{C}_{12}^{CPU} & = & \mathit{true} & \mathcal{C}_{21}^{CPU} & = & \mathit{false} \\
\mathcal{C}_{13}^{CPU} & = & \neg \mathit{blocking}_{31} & \mathcal{C}_{31}^{CPU} & = & \mathit{false} \\
\mathcal{C}_{23}^{CPU} & = & \neg \mathit{blocking}_{31} & \mathcal{C}_{32}^{CPU} & = & \mathit{blocking}_{31} \ ,
\end{array}$$

and we obtain the following priority functions implementing the admission control, and conflict resolution policies:

$$\begin{array}{rcl}
pr_{\mathsf{adm}}^{r} & = & \{(\mathit{blocking}_{13},\ BGN_{3,r} \prec_\infty U_{1,r}\circ)\} \oplus \\
& & \{(\mathit{blocking}_{13} \ \wedge\ \mathit{susp}_{1,CPU},\ BGN_{3,r} \prec_\infty rs_{1,CPU})\} \oplus \\
& & \{(\mathit{blocking}_{31},\ BGN_{1,r} \prec_\infty U_{3,r}\circ)\} \oplus \\
& & \{(\mathit{blocking}_{31} \ \wedge\ \mathit{susp}_{3,CPU},\ BGN_{1,r} \prec_\infty rs_{3,CPU})\} \\
& = & \{((u_1 \vee u_1') \wedge wr_3,\ g_3 \prec_\infty \{rs_1, v_1\})\} \oplus \{(u_1' \wedge wr_3,\ g_3 \prec_\infty rs_1)\} \oplus \\
& & \{((u_3 \vee u_3') \wedge wr_1,\ g_1 \prec_\infty \{rs_3, v_3\})\} \oplus \{(u_3' \wedge wr_1,\ g_1 \prec_\infty rs_3)\} \\
& = & \{((u_1 \vee u_1') \wedge wr_3,\ g_3 \prec_\infty \{rs_1, v_1\}),\ ((u_3 \vee u_3') \wedge wr_1,\ g_1 \prec_\infty \{rs_3, v_3\})\} \ .
\end{array}$$

Since the priority function pr_{res}^{CPU} is somewhat lengthy, we only show how to compute it:

$$\begin{array}{rcl}
pr_{\mathsf{res}}^{CPU} & = & \displaystyle\bigoplus_{i,j,k \in \{1,2,3\}} \{(\mathcal{C}_{ij}^{CPU},\ BGN_{j,CPU} \prec_0 BGN_{i,CPU})\} \oplus \\
& & \qquad \{(\neg\mathcal{C}_{ji}^{CPU} \wedge \mathit{active}_{i,CPU},\ PMT_{j,i}^{CPU} \prec_\infty U_{i,CPU}\circ)\} \oplus \\
& & \qquad \{(\mathcal{C}_{ij}^{CPU} \wedge \mathcal{C}_{ik}^{CPU} \wedge \mathcal{C}_{jk}^{CPU} \wedge \mathit{active}_{k,CPU},\ PMT_{j,k}^{CPU} \prec_0 PMT_{i,k}^{CPU})\} \ .
\end{array}$$

The priority function pr_{res}^{CPU} is well-defined: the action sets BGN, PMT, and $U\circ$ are pairwise disjoint, and the only action sets that are both dominating and dominated, are the BGN and PMT actions. It can easily be verified that the predicates $\mathcal{C}$ form a strict order, that is, for any $i, j \in \{1,2,3\}$, $\mathcal{C}_{ij}^{CPU} \ \wedge\ \mathcal{C}_{jk}^{CPU} \implies \mathcal{C}_{ik}^{CPU}$ (transitivity), and $\neg\mathcal{C}_{ii}^{CPU}$ (irreflexivity). Therefore, both the first and the third term of the priority function define a strict order on the BGN and PMT actions, respectively.

As there is only one non-preemptable resource r, pr_{adm}^{r} only expresses mutual exclusion on r, according to (PCP-2). The priority rule defined by the state constraints $\mathcal{C}_{ij}^{CPU}$ gives priority to the process with lower index, unless it blocks on r. Notice in particular $\mathcal{C}_{32}^{CPU}$: P_3 has priority over P_2 if P_3 is using r, and P_1 is waiting for r, so as to prevent priority inversion.

Properties

We show some important properties of our model of the priority ceiling protocol. We suppose that no process deadlocks due to its timing constraints while using some resource.

Theorem 5.1 (Unique blocker) *pr_{adm} prevents multiple blocking, in the sense that*

$$\forall i, j, k \,.\, k \leqslant \min\{i, j\} \,.\, \mathit{blocking}_{ik} \wedge \mathit{blocking}_{jk} \implies i = j$$

is an inductive invariant under the admission control policy defined by blocking: any process has at most one blocker of equal or lower priority.

Proof. Suppose that $\mathit{blocking}_{ik} \wedge \mathit{blocking}_{jk}$. Thus, $\exists r_1, r_2 \in R \,.\, U_{i,r_1} \wedge U_{j,r_2}$ with $\max\{c_{r_1}, c_{r_2}\} \leqslant k \leqslant \min\{i, j\}$. According to (5.1), this implies $i = j$. In other words, P_i is the only blocker of P_k. ■

Corollary 5.1 (Mutual exclusion) *Mutual exclusion on the resources follows from (5.1) with $r_1 = r_2$.*

By *transitive blocking* we understand that some process that is blocking a higher-priority process, is blocked itself.

Theorem 5.2 (Absence of transitive blocking) *The admission control policy defined by blocking preserves the inductive invariant*

$$\forall i, j, k \,.\, j < i \;:\; \neg(\mathit{blocking}_{ij} \;\wedge\; \mathit{blocking}_{jk}) \,.$$

Proof. Suppose that $\mathit{blocking}_{ij} \;\wedge\; \mathit{blocking}_{jk}$. Thus, P_i holds a resource r with $c_r \leqslant j < i$, and P_j holds another resource, which is in contradiction to (5.1). Therefore, there is no transitive blocking. ■

Theorem 5.3 (Deadlock-freedom) *The invariant (5.1) guarantees the relation blocking to be acyclic.*

Proof. Suppose $(P_{i_0}, \ldots, P_{i_{m-1}})$ to be a circuit in the waiting graph defined by *blocking*, that is, $\mathit{blocking}_{i_k, i_{(k+1)\ mod\ m}}$ for any $k \in \{0, \ldots, m-1\}$, where P_{i_0} has minimal fixed priority among the processes which are vertices of the circuit. Hence, P_{i_0} holds a resource r_1 that $P_{i_{m-1}}$ is waiting for. $P_{i_{m-1}}$, in turn, holds another awaited resource, r_2. Formally,

$$\exists r_1, r_2 \in R \,.\, U_{i_0, r_1}(s) \;\wedge\; c_{r_1} \leqslant i_{m-1} \leqslant i_0 \;\wedge\; U_{i_{m-1}, r_2}(s)$$

which is in contradiction to the invariant (5.1). ■

Let inv be the state constraint characterizing the states s where priorities are inversed, in the sense that a high-priority process P_k is blocked by some process P_i, which, in turn, is preempted by a third process P_j of lower priority than P_k. Formally,

$$inv(s) = \exists i, j, k \;.\; i \neq j \;\wedge\; k < j \;\wedge\; blocking_{ik}(s) \;\wedge\; U_{j,CPU}(s) \;.$$

Theorem 5.4 (Absence of priority inversion) *The timed system* $(TS,\ pr_{\mathsf{sync}} \oplus pr_{\mathsf{adm}} \oplus pr_{\mathsf{res}})$ *is free of priority inversion, in the sense that* $\neg inv$ *is an inductive invariant.*

Proof. Consider some control state s of TS such that $\neg inv(s)$, that is,

$$\forall i, j, k \;.\; i \neq j \;\wedge\; k < j \;.\; \neg(blocking_{ik}(s) \;\wedge\; U_{j,CPU}(s)) \;.$$

We show that for any transition (s, a, s') of $(TS,\ pr_{\mathsf{sync}} \oplus pr_{\mathsf{adm}} \oplus pr_{\mathsf{res}})$, $\neg inv(s')$. We distinguish three cases:

- If $\forall i, j, k \;.\; i \neq j \;\wedge\; k < j \;.\; \neg blocking_{ik}(s) \;\wedge\; \neg U_{j,CPU}(s)$, then it is impossible, for any i, j, k with $i \neq j \;\wedge k < j$, to make inv true in one transition, due to the requirements $BGN_{i,r} \subseteq BGN_{i,CPU}$, and $\bullet W_{k,r} \subseteq U_{k,CPU^{\circ}}$.

- If $\exists i, k \;.\; blocking_{ik}(s)$, then for any process P_j, $j \neq i \;\wedge\; k < j$, that is waiting for some resource, we have also $blocking_{ij}(s)$ according to the definition of $blocking$, and $\mathcal{C}_{ij}^{CPU}(s) = i < j \;\wedge\; \forall l \leqslant i \;.\; \neg blocking_{jl}(s) \;\vee\; j < i$. If $j < i$, then $\mathcal{C}_{ij}^{CPU}$ holds. If, on the other hand, $i < j$, suppose that $\exists l \leqslant i \;.\; blocking_{jl}(s)$. Thus, P_j uses some resource $r \in R_n$ such that $c_r \leqslant l \leqslant i < j$, which is in contradiction with invariant (5.1). In both cases, $\mathcal{C}_{ij}^{CPU}$ holds, and pr_{res} prevents P_j from getting the CPU.

- If $\exists j \;.\; U_{j,CPU}(s)$, then no process P_k, $k \neq j$, can enter a waiting state and make $blocking_{ik}$ become true, according to the requirement $\bullet W_{k,r} \subseteq U_{k,CPU^{\circ}}$.

  If some process P_i, $i \neq j$, acquires a resource $r \in R_n$, then it has to get the CPU at the same time, according to the requirement $BGN_{i,r} \subseteq BGN_{i,CPU}$. This is only possible if P_i can preempt P_j, that is, if $\mathcal{C}_{ij}^{CPU}(s)$. Together with the hypothesis that $\forall k < j \;.\; \neg blocking_{ik}(s)$, it follows from the definition of $\mathcal{C}_{ij}^{CPU}$ that $\forall k < i \;.\; \neg blocking_{jk}(s)$. Therefore, we also have $\forall k < i \;.\; \neg blocking_{jk}(s')$.

Therefore, any successor state s' of s satisfies $\neg inv(s')$. ∎

Properties such as worst-case blocking time and schedulability are the same as for the priority ceiling protocol and can be proven as in [SRL90]. Notice that we do not require that the resource allocations and releases are "properly nested"; they can be overlapping.

Our model describes a set of processes scheduled under the priority ceiling protocol by avoiding any priority inheritance mechanism. We believe that this improves understandability of the model.

The Priority Ceiling Protocol with EDF

Instead of basing the priority ceiling protocol on fixed process priorities, the latter can be dynamic, for example according to the EDF policy, as proposed in [CT94]. Using EDF priorities in the priority ceiling protocol allows to take advantage of the optimality of EDF among the dynamic-priority scheduling policies [LL73], which means in particular that the EDF policy can be feasible for systems where fixed-priority scheduling is not feasible.

In order to determine the process priorities according to EDF rather than according to the process index, replace the state constraints

$$\begin{aligned} \mathcal{C}_{ij}^{CPU} \quad \Longleftrightarrow \quad & i < j \ \wedge \ \forall k \leqslant i \,.\, \neg \mathit{blocking}_{jk} \ \vee \\ & j < i \ \wedge \ \exists k < j \,.\, \mathit{blocking}_{ik} \end{aligned}$$

in our model of the priority ceiling protocol by

$$\begin{aligned} \mathcal{C}_{ij}^{CPU} \quad \Longleftrightarrow \quad & \pi_{ij} \ \wedge \ \forall k \leqslant i \,.\, \neg \mathit{blocking}_{jk} \ \vee \\ & \neg\pi_{ij} \ \wedge \ \exists k < j \,.\, \mathit{blocking}_{ik} \ , \end{aligned}$$

where $\pi_{ij} = D_{i,CPU} - t_i < D_{j,CPU} - t_j \ \vee \ D_{i,CPU} - t_i = D_{j,CPU} - t_j \ \wedge i < j$ is the state constraint expressing the priority of P_i over P_j according to EDF.

5.5 The Scheduled System

The obtained timed system with priorities $(\mathit{TS},\ pr_{\mathsf{sync}} \oplus pr_{\mathsf{pol}})$, with

$$pr_{\mathsf{pol}} \quad = \quad \bigoplus_{r \in R} (pr^r_{\mathsf{adm}} \ \oplus \ pr^r_{\mathsf{res}}),$$

represents the synchronized system $(\mathit{TS}, pr_{\mathsf{sync}})$, scheduled under the policy modelled by pr_{pol}. Notice that the complexity of the priority function $pr_{\mathsf{sync}} \oplus pr_{\mathsf{pol}}$ essentially depends on the complexity of inter-process coordination, not that of the process models.

According to chapter 3, all timing constraints, including responsiveness (bounds on response times), end-to-end timing constraints, inter-arrival times, execution times, and deadlines, are encoded in the guards of the process models of which TS is composed. Feasibility of the scheduling policy pr_{pol} means deadlock-freedom of all processes — and liveness in the case of structurally live processes — in the scheduled system $(\mathit{TS},\ pr_{\mathsf{sync}} \oplus pr_{\mathsf{pol}})$, which implies that all timing constraints are met.

There are several approaches to verify that all processes are live in $(\mathit{TS},\ pr_{\mathsf{sync}} \oplus pr_{\mathsf{pol}})$: results of scheduling theory where they are applicable, compositional reasoning about structural properties guaranteeing liveness by construction as in chapter 4, model-checking and synthesis [Tri98, Pet99, AGP+99, TA99], checking for bisimulation with live processes [KLP+98, BACC+98] using tool sets such as CADP [FGK+96] or PARAGON [BALS99], and estimation of worst-case completion times (see section 5.6).

In order to express feasibility of the scheduling policy pr_{pol} as an X-constraint that can be model-checked or synthesized, we define for each application process P_i the *scheduling requirement*

$$K^i_{\mathsf{sched}} = \bigvee_{s=(s_1,\ldots,s_n)\in S} s_i \wedge \Diamond^{\mathbf{b}_s} guards_{s_i}(\mathbf{x})$$

expressing the possibility of P_i to eventually execute an action in $TS = (S, A, T, X, \mathbf{b}, h)$, and thus the absence of deadlock. If the conjunction $K_{\mathsf{sched}} = \bigwedge_{i\in\{1,\ldots,n\}} K^i_{\mathsf{sched}}$ is an inductive invariant of the scheduled system $(TS,\ pr_{\mathsf{sync}} \oplus pr_{\mathsf{pol}})$, then the scheduling policy is feasible, and the timing constraints are guaranteed to be met [AGS00, AGS01].

Example 5.4 *The scheduling requirement of the sporadic process* `P` *of figure 2.2 is*

$$K^{\mathsf{P}}_{\mathsf{sched}} = \mathit{sleeping} \vee \mathit{waiting} \vee \mathit{computing} \wedge x_p \leqslant 3 \wedge t_p \leqslant 5 \wedge t_p - x_p \leqslant 3\ .$$

If `P` *is composed with other processes in a scheduled system, and the system is initialized in* $K^{\mathsf{P}}_{\mathsf{sched}}$*, then some action of* `P` *will always eventually become enabled and the timing constraints are respected, provided that* $K^{\mathsf{P}}_{\mathsf{sched}}$ *is an inductive invariant of the scheduled system.*

In the scheduled system, synchronization between processes, and the scheduler are represented by the priority functions pr_{sync} and pr_{pol}, respectively. Both the synchronization layer and the scheduler observe the system state — or an abstraction of it that must be sufficiently fine to distinguish the truth value of the conditions of pr_{sync} and pr_{pol} — and disable actions according to the priority order applying to the current system state. Priorities are a means to restrict the system behaviour dynamically. However, modelling coordination between the processes as a priority function does not anticipate an implementation choice. In particular, it does not influence the decision how the synchronization layer is implemented, nor whether the system runs under dynamic or pre-runtime scheduling [XP93, XP00]. The priority function can be used to construct a pre-runtime scheduler, and it can make up the heart of a runtime scheduler.

5.6 Model-Based Timing Analysis

Disposing of a description of nearly all aspects of coordination between the processes — mutual exclusion, atomicity, resource allocation and scheduling policies — in the form of a priority function, it seems interesting to take advantage of that wealth of information in order to estimate upper bounds of the blocking times imposed by the priority function, in order to derive a diagnostic about the liveness — or schedulability — of the processes in the global system model.

For any process P_i modelled by the timed system TS_i^{PMTN}, let $T_i = T_i^l$ denote its minimal inter-arrival time, and let E_i be its maximal execution time in absence of interference with other processes. Let (TS, pr) be the scheduled system of processes to be analysed, where $TS = (S, A, T, X, \mathbf{b}, h)$, and $pr = pr_{\mathsf{sync}} \oplus pr_{\mathsf{pol}}$. Since the modelling framework presented in chapters 3 and 5 is very general, we make some hypotheses about the structure of the model of the real-time application to be scheduled, in order to simplify the timing analysis:

- There is no inter-process synchronization, apart from Σ_{PMTN} modelling preemption.
- The processes have no deadline constraints apart from the fact that execution must terminate before the next arrival of the process.
- (TS, pr) is structurally safe with respect to the set S' of control states where pr is well-defined, that is, where $\forall (s, \mathbf{x}) \in S' \times \mathbb{R}^m_{\geqslant 0} \,.\, pr(s, \mathbf{x}) \neq \bot$.

The idea behind the analysis method is similar to the critical instant/busy-period analysis used in fixed-priority schedulability analysis, such as rate-monotonic analysis [LL73, LSD89, HKO+93]: find, for each process P_i, an upper bound C_i of its worst-case completion time (WCCT) — that is, the time elapsed from its arrival until the end of its execution, including possible interference of other processes —, and determine whether the process will meet its deadline constraint in the worst case.

An upper bound C_i for the completion time of P_i, that is, the sum of its execution time and the time during which P_i can be blocked by other processes, is given by the solution of the equation

$$x \quad = \quad E_i + \sum_{j \in \{1,\ldots,n\} \setminus \{i\}} \left\lceil \frac{x}{T_j} \right\rceil \times B_{ji}$$

where B_{ji} is the maximal time process P_j spends — without being blocked by other processes — in the set of states in which it may interfere with P_i:

$$B_{ji} \quad = \quad \mathit{max\_staytime}_j(\mathit{interferring\_states}_{ji}) \;.$$

Intuitively, C_i is obtained as the execution time E_i, plus the time process P_i may be blocked by other processes during its completion time C_i. Notice that in states where P_j is blocking P_i, and P_i is blocked itself by another process P_k, transitivity of the priority order makes sure that the indirect blocking of P_i by P_k is taken into account. The set $\mathit{interferring\_states}_{ij}$ is defined as the set of states of process P_j for which there is some state $(s, \mathbf{x})$ of TS such that some action of P_j dominates some action of P_i in $pr(s, \mathbf{x})$:

$$\begin{aligned}\mathit{interferring\_states}_{ji} \quad = \quad & \big\{ s_j \mid \exists s = (s_1, \ldots, s_n) \in S' \quad \exists \mathbf{x} \in \mathbb{D}^m \\ & \exists a_i \in \{s_i\}\circ \quad \exists a_j \in \{s_j\}\circ \;.\; \; a_i \prec_0 a_j \in pr(s, \mathbf{x}) \big\} \;.\end{aligned}$$

Since the set of states $\mathit{interferring\_states}_{ji}$ can be entered and left more than once — as in the example of section 7.1 —, $\mathit{max\_staytime}_j(\mathit{interferring\_states}_{ji})$ can be over-estimated as the sum of the $\mathit{max\_staytime}_j$ in the weakly connected components of $\mathit{interferring\_states}_{ji}$ in the untimed transition structure of TS_j^{PMTN}. Let C be the set of weakly connected components of $\mathit{interferring\_states}_{ji}$ in the untimed transition structure of TS_j. Then,

$$\mathit{max\_staytime}_j(\mathit{interferring\_states}_{ji}) \quad = \quad \sum_{c \in C} \mathit{max\_staytime}_j(c) \;.$$

Let

$$R_i = \frac{T_i}{C_i}$$

denote the ratio between the minimal inter-arrival time of process P_i, and its maximal completion time C_i. Then, process P_i is deadlock-free in (TS, pr) if the ratio is greater or equal to one for all processes, that is, if the set of processes is schedulable under the constraints imposed by the priority function pr. If pr is such that no process can deadlock in a state where it is blocking another process, then P_i is deadlock-free in (TS, pr) if $R_i \geqslant 1$.

In the computation of R_i, we have made no distinction whether the used resources are preemptable or not, since the necessary information is provided by the priority function pr. If the CPU is not preemptable, then P_i will get blocked due to the CPU in states $W_{i,CPU}$, but not in $U_{i,CPU}$ (unless there are further priorities ensuring for example mutual exclusion on another resource). If the CPU is preemptable and allocated according to some time-independent scheduling policy such as RMS, then a waiting process will only be blocked by higher-priority processes in states $W_{i,CPU}$ and $susp_{i,CPU}$.

Example 5.5 *Taking up our favourite two-process example 3.8 with timing constraints as in figure 4.3, let us analyse the timing of both processes in the synchronized system* $(\|_{\emptyset}\{TS_1, TS_2\}, pr_{mutex} \oplus pr_{atom})$. *Structural safety with respect to the states in*

$$S' = \{s_1, w_1, u_1^1, u_1^{12}\} \times \{s_2, w_2, u_2^2, u_2^{12}\} \setminus \{(u_1^1, u_2^2), (u_1^1, u_2^{12}), (u_1^{12}, u_2^2), (u_1^{12}, u_2^{12})\}$$

has been shown in example 4.7.

The maximal execution times of both processes in isolation are $E_1 = 6$, *and* $E_2 = 3$, *the minimal inter-arrival times are* $T_1 = 10$, *and* $T_2 = 12$. *According to the priority function*

$$\begin{aligned} pr_{mutex} \oplus pr_{atom} = \; & \big\{ \big(u_1^1 \wedge u_2^{12},\; bgn_2^2 \prec_\infty bgn_1^2 \prec_\infty \{bgn_2^1, end_2\}\big), \\ & \big(u_1^{12} \wedge u_2^2,\; bgn_1^1 \prec_\infty bgn_2^1 \prec_\infty \{bgn_1^2, end_1\}\big), \\ & \big(u_1^1 \wedge \neg(u_2^2 \vee u_2^{12}),\; bgn_2^2 \prec_\infty bgn_1^2\big), \\ & \big(u_2^2 \wedge \neg(u_1^1 \vee u_1^{12}),\; bgn_1^1 \prec_\infty bgn_2^1\big), \\ & \big(u_2^{12} \wedge w_1,\; bgn_1^1 \prec_\infty end_2\big),\; \big(u_1^{12} \wedge w_2,\; bgn_2^2 \prec_\infty end_1\big), \\ & \big(u_1^1 \wedge u_2^2,\; \perp\big)\big\}\;, \end{aligned}$$

we deduce that

$$\begin{aligned} interferring\_states_{12} &= \{u_1^1, u_1^{12}\} \\ interferring\_states_{21} &= \{u_2^2, u_2^{12}\}\;, \end{aligned}$$

and obtain by analysing the processes separately, $B_{12} = 6$, *and* $B_{21} = 3$, *and by solving the recursive equation, the worst-case completion times* $C_1 = C_2 = 9$. *Since both ratios* $R_1 = T_1/C_1 = 10/9$ *and* $R_2 = T_2/C_2 = 12/9$ *are greater than one, the synchronized system is schedulable.*

The method presented in this section is a simple estimation to illustrate how the priority function pr, which carries information about most aspects of process interaction, can be used for timing analysis. As this topic is not within the scope of this thesis, the possibility of model-based timing analysis has not been investigated in depth. However, its seems possible and promising to define less pessimistic timing analysis methods, for example by taking advantage of time-dependent priorities generated by scheduling policies such as EDF, or by analysing the priority function not for pairs of processes, in order to preserve more information about the interaction between all processes.

Qualitative timing information about finite-state systems can be obtained using the VERUS tool [CCMM95], but this approach based on symbolic model-checking suffers from state space explosion. Other generalizations of the worst-case completion time calculus under rate-monotonic scheduling have been proposed in [Ves94, Bin97, Ves00] taking into account blocking on non-preemptable resources and providing quantitative results in the form of critical scaling factors by which the execution times of the processes can be multiplied while still keeping the system schedulable. [HKL94] allows tasks to consist of subtasks having their own execution time, fixed priority, and deadline. [SK00] discusses the application of schedulability analysis to object-oriented models with message passing, in the case of fixed-priority and PCP scheduling.

5.7 Discussion

In chapters 3 and 5, we have presented a general modelling framework for reactive real-time systems. Different aspects of coordination between processes are modelled separately, and composed to obtain a model of the scheduled system. The composed priority function describes the overall restriction of the system behaviour ensuring the required properties. It carries information about how the scheduled system may behave under different conditions, allowing to answer questions such as "what are the system states in which process P_i has priority over process P_j as to acquiring resource r?". In other words, the information provided by priority functions helps to improve predictability of the modelled system.

This modular design principle is supported by the compositionality results of chapter 4, which allow reasoning about safety, liveness, and timing properties without explicitly constructing the product of the timed systems modelling the processes.

Our results suggest the following iterative design process:

1. modelling the application processes as structurally live timed systems;
2. construction of a structurally live synchronized system by adding the synchronization layer;
3. checking it for structural safety. In the negative case, there are two possibilities:
   - running the priority synthesis algorithm 4.1, or
   - re-designing the synchronized system by taking into account the obtained diagnostic;

4. adding the scheduler;

5. checking the scheduled system for feasibility of the scheduling policy, using the worst-case completion time calculus as a first feasibility test. Since this test is conservative, it may be necessary to use an exact verification algorithm in order to verify or refute schedulability.

   In the negative case, the same possibilities as above apply, that is, priority synthesis, or refinement of the scheduler. Guidance for choosing pr_{pol} can be provided by the diagnostics of PROMETHEUS, or the used verification tool. The idea is to refine pr_{pol} so as to eliminate runs of the system that leave K_{sched}, that is, lead to the deadlock of some process.

This incremental modelling method, based on a well-founded formal framework, relies on structural properties that are inexpensive to check. Model-checking may become necessary only in last step, where many properties have already been ensured on the scheduled system. We illustrate the use of this incremental process in the case studies presented in chapter 7.

The use of scheduling policies can make schedulability analysis more tractable. The application of a scheduling policy pr_{pol} to a synchronized system (TS, pr_{sync}) reduces the non-determinism of the latter, and consequently the complexity of its state space. However, pr_{pol} should be chosen so that $(TS,\ pr_{\mathsf{sync}} \oplus pr_{\mathsf{pol}})$ has a non empty invariant implying K_{sched}, and this cannot be guaranteed by existing results.

It is possible to model further properties in our framework, such as context switching overhead that is to be systematically taken into account, interrupts, or fault models. For example, resources that may fail, can be specified by using maximal-priority fault processes that can occupy the resource, until the error is detected and recovered, as in [JJ01].

Chapter 6

The Modelling Tool PROMETHEUS

6.1 Motivation

PROMETHEUS [1] is a modelling tool allowing the user to specify and compose real-time systems, by means of synchronization and priorization of actions with a scheduler specified in a high-level description language [Gößl01]. The resulting model is checked for safety and liveness properties, which can — up to a certain degree — be guaranteed by PROMETHEUS. The composed system can be output in several formats.

Initially, PROMETHEUS was intended for validation of the theoretical results discussed in the previous chapters. Therefore, it implements most of these concepts, including

- Petri nets with urgency, timed systems with priorities, and parallel composition as the basic modelling formalism;
- the automatic generation of preemption and of the priority function for mutual exclusion;
- analysis of the structural liveness and other timing properties of the components, and generation of a diagnostic of the structural liveness and safety of the compound system; and
- the possibility of specifying a scheduler by defining the admission control, and conflict resolution policies.

During the work on PROMETHEUS and its testing on small and medium-size examples, certain aspects of the specification task turned out to be tedious and gave the idea of hiding technical details, and adding functionality to automatize the task. For example, describing the priority ceiling protocol on the action level led even for

[1] Prometheus, nephew of Kronos, had the power to foresee the future. PROMETHEUS aims at making real-time systems predictable.

small systems to a number of priorities that was hard to debug and to modify by hand. This problem led to the idea of a high-level scheduler description allowing abstraction from the actions, and quantification over resources and processes. Similarly, process templates were introduced to replace multiple occurrences of a process type with the same untimed transition structure, but different timing constraints. The possibility of instantiating scheduler templates was added to overcome the need to adapt the description of some scheduling policy each time it is applied to a new system of processes; it allows to establish a library of frequently used schedulers such as RMS, EDF or PCP. The notion of templates is similar to the idea of *specification patterns* [BALS99], providing reusable pieces of specification.

As the examples became somewhat larger, debugging of the specifications was of growing importance. Therefore, detailed diagnostics, mainly about the component systems and possible circuits in the priority function, were added, in order to accelerate debugging, but also to gain confidence in the correctness of the specification. With the connection of PROMETHEUS to the validation platform IF [BFG+00, Nak01], efficiency of the generated model became crucial, in order to tackle non-trivial case studies.

Figure 6.1 gives an overview of the context of PROMETHEUS, and the toolset connected to the platform IF. The validation environment IF, which has been developed

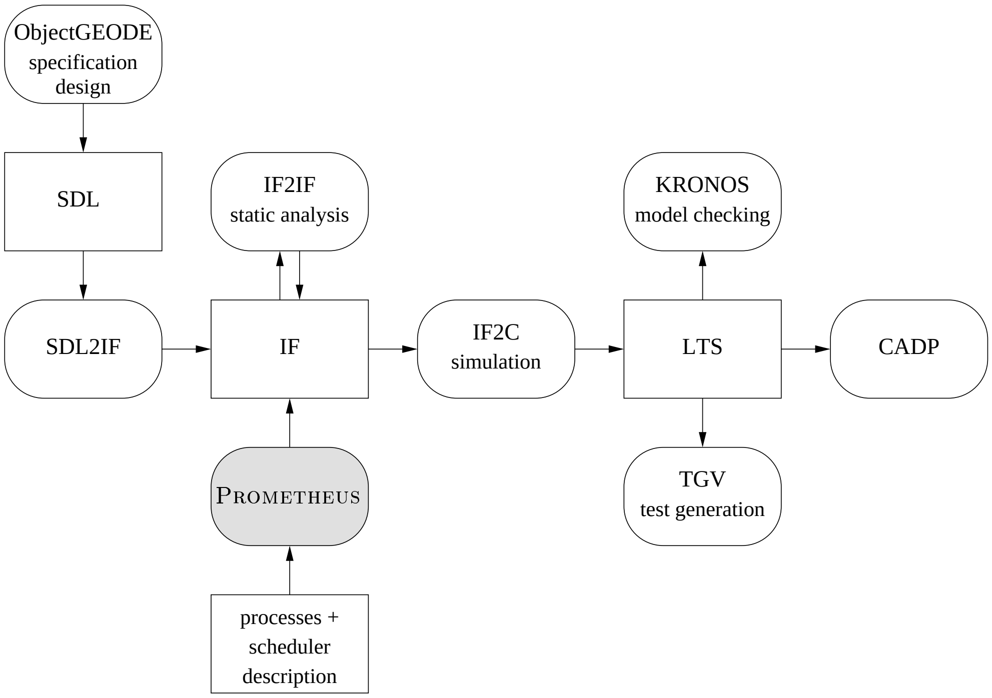


Figure 6.1: The context of PROMETHEUS.

at VERIMAG, is based on an *intermediate format* for timed asynchronous systems, the IF language, and integrates tools operating on different levels of abstraction.

At the *specification level*, systems can be described in a high-level description language such as SDL [IT99]. The translator SDL2IF allows an automatic translation to the IF language.

At the *intermediate level*, the system is represented as a set of parallel communicating processes. At this level, the optimization tool IF2IF based on static analysis techniques allows transformations such as clock reduction, and program slicing [Wei84].

The tool IF2C generates a simulator that uses techniques such as partial order reduction and on-the-fly model-checking to explore the state space of the IF specification, giving access at the *semantic level*, to the corresponding labelled transition system (LTS). The latter can be analysed using the tool suite CADP [FGK+96], including the minimization and comparison tool ALDEBARAN based on bisimulation, and the alternating-free μ-calculus model-checker EVALUATOR. At the same level are also available the TCTL model-checker KRONOS [DOTY96, Yov97], and the test generator TGV [FJJV97].

PROMETHEUS consists of 20k lines of C++ in 15 classes, and three parsers for the input files. It uses the STL (Standard Template Library), IF, and Cudd libraries. Although still a prototype, PROMETHEUS is a useful and efficient support to automatize tedious and error-prone tasks in modelling real-time applications. Its connection to the platform IF allows the user to take advantage of the existing toolset connected to IF, in order to perform static analysis, simulation, model-checking, or test generation on the IF specification generated by PROMETHEUS. The independence between the scheduler description and the specification of the system of processes to be scheduled, introduces a great deal of flexibility, since any of them can be modified independently. This allows, for example, to refine a scheduler, according to the diagnostic provided by PROMETHEUS and other tools, until it ensures liveness of the processes, without the need of modifying the latter.

However, PROMETHEUS is still a prototype: in the current version, not all modelling primitives can be translated into an IF specification — as for example the use of preemptable resources — although there is no technical reason for this restriction. The integration of PROMETHEUS into the IF platform is one-way, since the input language is proper to PROMETHEUS. Accepting system descriptions in SDL or IF would further increase the usefulness of PROMETHEUS. Finally, automatic code generation would be interesting to allow code generation from the same specification that has been verified, simulated, or tested.

6.2 The Architecture of PROMETHEUS

Figure 6.2 shows the global architecture of PROMETHEUS. As an input, PROMETHEUS reads a file describing the global system configuration including the process declarations, available resources, synchronizing actions, and optionally the name of a file specifying the scheduler to be used. The processes can be specified directly in the system file, or instantiated from process templates that are automatically included. From this description, PROMETHEUS generates an IF specification of the processes,

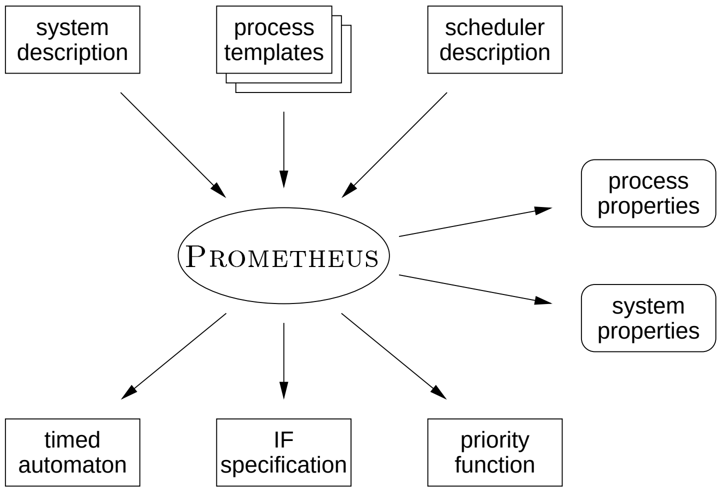


Figure 6.2: Overview of the tool PROMETHEUS.

the priority function describing the coordination between the processes, including the scheduling policy, and optionally a timed automaton representing the product timed system with priorities.

The priority function is the main part of the system model, since it models coordination between the processes, and its complexity increases with growing complexity of the coordination. Therefore, its representation is crucial for the efficiency of PROMETHEUS. Remember that any element of the priority function is a pair composed of a state constraint, and a priority order. The state constraints are represented symbolically, whereas the associated priority orders, generally less critical, are represented enumeratively.

6.3 Using PROMETHEUS

PROMETHEUS is a command-line tool that is invoked as follows:

`prometheus` [<options>] <system>

Possible options are:

-h to get help

-u to apply the priority synthesis algorithm 4.1 to try and make bad states unreachable

-d to make the system deterministic. With this option, the scheduled system is made deterministic by completing the priority order associated with any system state to a total order on all controllable actions. If moreover all controllable actions are specified as eager — thus eliminating non-deterministic timing —, the only remaining non-determinism comes from uncontrollable actions. This option is motivated by the fact that the determinized system is generally simpler than the original one. Besides, determinism is sometimes required for embedded safety-critical systems that have to be certified.

-a to output the product timed system

-i to create a timed automaton with invariants rather than a timed system.

-j same as -i, but generate convex invariants.

-r discard states that are unreachable from the initial state of the Petri net. This option is ignored if none of the options **-i**, **-j** is specified.

-v verbose mode.

The grammar of the system, process template, and scheduler description files are given in appendix B.3.

6.3.1 The System Description

This file declares the different components of the system, as well as some high-level aspects of their interaction. It essentially consists of the following parts (optional parts are written in italic):

Resources The resources available in the systems. Resources are declared either preemptable or non-preemptable.

Processes The components (processes). Each process can be described directly — see section 6.3.2 —, or by giving the filename of a template and instantiating processes that can be parameterized by a list of integers. By convention, the first integer specifies the fixed priority π of the process. If the list of parameters is empty, then the fixed priority is set to zero. The fixed priority has the following effects:

- A process P_i can preempt a process P_j using a preemptable resource only if $\pi(P_i) \geqslant \pi(P_j)$. Therefore, if a system is to be scheduled according to some preemptive dynamic-priority algorithm such as EDF, all processes must have the same fixed priority.
- In the scheduler description, the fixed priority of any process P can be referred to via the function **prio(P)**.
- The *priority ceiling* of any non-preemptable resource r is defined as the maximal fixed priority of all processes having at least one place where r is used. This value can be used by the scheduler via the function **ceiling(r)**.
- When determinizing the system using the option **-d**, the interleaving actions of a higher-priority process are given priority over the interleaving actions of a lower-priority process, unless specified differently by the priority function modelling coordination between the processes.

Synchronization A list of synchronization actions, each consisting of the synchronizing actions and a synchronization mode as in section 2.4.4. An exclamation mark "!" behind an action means that interleaving transitions labelled by this action are pruned off. In this case, synchronization is blocking for that action. By default, both actions interleave, according to the non-blocking maximal progress semantics. A hierarchical synchronization of actions (that is, a synchronization in which synchronized actions take part) is possible. At least one of the synchronizing actions must be controllable.

Scheduler The filename of a scheduler description file (see section 6.3.3). Optionally, integer parameters can be passed.

Assertion is a boolean expression on control states and variables. Its utility is twofold: first, if the product timed system is constructed, then the assertion is checked in each control state. A warning message is issued if a control state does not satisfy the assertion, and transitions leaving this state are not explored. Second, if the assertion expression contains only control states, then the states violating the assertion are considered as "bad" states to made unreachable if the option **-u** is selected.

System 6.1 shows a system description declaring one non-preemptable resource `cpu`, and instantiating three periodic processes that are scheduled under a scheduler called `edf.sched`.

System 6.1 Three periodic processes, scheduled under EDF.

```
SYSTEM edf_test
        NONPREEMPTABLE cpu;

        periodic.proc
                p1 (0, 2, 5),   // fixed priority, exec. time, period
                p2 (0, 3, 11),
                p3 (0, 4, 17)

        SCHEDULER edf.sched()
END edf_test
```

6.3.2 The Process Description

The description of a process can be part of the system description file, the scheduler description file, or — with slight syntactical differences — make a file of its own. A process description essentially defines a Petri net with urgency, and has the following structure:

Parameters *(in template files only):* a list of parameters that are instantiated in the system description file. If there is at least one parameter, then the first parameter is interpreted as the fixed process priority, otherwise the priority is supposed to be 0. In the remainder of the process description, a parameter may replace any integer expression. Parameters can be referenced by a leading dollar sign (for example, **$period** for the parameter *period*).

Places The set of places of the Petri net.

Resources A list of resource identifiers, each identifier followed by a subset of the above places where the resource is in use.

Clocks The set of clocks of the Petri net.

Variables A set of integer variables. They are for the convenience of specification only.

Guards In this section, a set of guards can be defined, especially guards that are shared by several transitions.

Initial marking of the Petri net, specified by the set of initially marked places.

Transitions the set of transitions of the Petri net. A transition is described by the following items:

1. the name of the action by which it is labelled;
2. its controllability, which is *controllable* by default;
3. its input and output places;
4. its guard;
5. the set of clocks to be reset; and
6. an optional assignment to a variable. The left-hand side can be any variable of the system that does not need to belong to the same process. The right-hand side is restricted to expressions on parameters and variables of the process to which the transition belongs, and constant expressions.

Process 6.1 shows the process template `periodic.proc` included by system 6.1. Transitions `arrive` and `rl` are uncontrollable, whereas transition `bgn` acquiring the resource `cpu` is controllable.

Process 6.1 Template `periodic.proc` specifying a simple periodic process.

```
PND periodic (prio, E, T)
        PLACES sleep wait use
        RESOURCE cpu: use
        CLOCKS t x
        INIT: sleep;

        arrive: UNCTRL
                IN: sleep
                OUT: wait
                GUARD t = $T DELAYABLE
                RESETS t;

        bgn:    IN: wait
                OUT: use
                GUARD TRUE EAGER
                RESETS x;

        rl:     UNCTRL
                IN: use
                OUT: sleep
                GUARD x = $E DELAYABLE;
END periodic
```

6.3.3 The Scheduler Description

This file serves to specify the admission control, and conflict resolution policies, see sections 5.2 and 5.3. More generally, any priority function can be specified in this file. It looks as follows:

Parameters A list of parameters that are instantiated in the system description file. In the remainder of the scheduler, a parameter may replace any integer expression. Parameters of processes can be referenced by a leading dollar sign, for example, **$P.period** for the parameter *period* of process P.

Petri net This optional Petri net with urgency may be used to define schedulers that have a memory, such as event handlers that keep trace of requests that have not been served yet.

Non-preemptable resources A list of non-preemptable resources that must be a subset of all resources declared in the system description file. In other words, the scheduler may treat resources as non-preemptable that are preemptable according to the system description. The user is informed of this redeclaration by a warning message.

Admission control The admission control policy is defined by state constraints $blocking_{ij}$ specifying when process P_i blocks process P_j on some non-preemptable resource, as discussed in section 5.2. The definition of *blocking* is of the form

`[FORALL pid`$_1$`: PROCESS] [FORALL pid`$_2$`: PROCESS]`
`BLOCKING (pid`$_1$`,pid`$_2$`) := state_constraint`

The process identifiers can be universally quantified. `state_constraint` can be prefixed by a list of existential and universal quantifications on the domains of processes, resources, preemptable resources, and non-preemptable resources. It may contain propositions on control states and clock states, as well as the pre-defined predicates **using(P,r)** and **waiting(P,r)**, standing for $U_{P,r}$ and $W_{P,r}$, and characterizing the control states where process P uses resource r, and process P waits for resource r, respectively. Moreover, integer expressions occurring in *blocking* may use the pre-defined expressions **prio(P)** for the fixed priority of some process P, and **ceiling(r)** for the priority ceiling of some resource r.

The predicate *blocking* is statically evaluated into a propositional state constraint. In the current version of PROMETHEUS, the admission control policy defined by *blocking* applies to all non-preemptable resources.

Priority function especially defining the conflict resolution policy. The priority function is given as a list of priority rules of the form

`aid`$_1$ $<k$ `aid`$_2$ [`IF state_constraint`]

where `aid`$_1$, `aid`$_2$ are action identifiers, or of the form

`[FORALL pid`$_1$`: PROCESS] [FORALL pid`$_2$`: PROCESS]`
`pid`$_1$ $<k$ `pid`$_2$ [`IF state_constraint`]

In the first case, the priority is assigned between the specified actions; in the second case, the priority applies to all actions of the specified processes which acquire some resource (preemptable or non-preemptable). The optional `state_constraint` specifies the system states where the priorities apply. It has the same structure as for *blocking*. In addition, it can contain the predicate *blocking*. If `state_constraint` is omitted, then it is assumed to be *true*.

Scheduler 6.1 shows the EDF scheduler used by system 6.1: for any two processes P_i, P_j, the actions of the latter acquiring some resource (in the case of system 6.1, the CPU) dominate the acquiring actions of the further if either P_j is closer to its deadline, or if both processes have the same distance to their deadlines, and the parameter `pid` of P_j is less than that of P_i.

Scheduler 6.2 specifies the priority ceiling protocol, as discussed in section 5.4.2. In the first part of the scheduler specification, the state constraint *blocking* is defined. Instead of assuming that processes of lower indexes have higher fixed priority, the fixed

Scheduler 6.1 Specification of an EDF scheduler.

```
SCHEDULER edf()
    PRIORITIES                      // conflict resolution
        FORALL Pi: PROCESS
        FORALL Pj: PROCESS
            Pi <0 Pj  IF  [ Pi.t - Pj.t < $Pi.T - $Pj.T ] OR
                          [ Pi.t - Pj.t = $Pi.T - $Pj.T ] AND
                                           $Pi.pid > $Pj.pid
END edf
```

priority of some process P_i is returned by the function `PRIO` (P_i). The second part specifies with two priority rules the conflict resolution policy. All three state predicates use quantifications.

Scheduler 6.2 Specification of a PCP scheduler.

```
SCHEDULER pcp()
    forall Pi: process
    forall Pj: process
        blocking (Pi, Pj) :=        // admission control
            exists r1: nonpreemptable
            exists r2: nonpreemptable
                USING (Pi, r1) and WAITING (Pj, r2) and
                CEILING (r1) >= PRIO (Pj)

    PRIORITIES                      // conflict resolution
        forall Pi: process
        forall Pj: process
            Pj <0 Pi  IF
                forall Pk: process
                    PRIO(Pi) > PRIO (Pj) and
                    (PRIO(Pk) < PRIO(Pi) or not BLOCKING (Pj, Pk))

        forall Pi: process
        forall Pj: process
            Pj <0 Pi  IF
                exists Pk: process
                    PRIO(Pi) < PRIO (Pj) and
                    PRIO(Pk) > PRIO(Pj) and BLOCKING(Pi, Pk)
END pcp
```

6.4 How Prometheus Works

Figure 6.3 gives an overview of the internal structure of Prometheus. On invocation

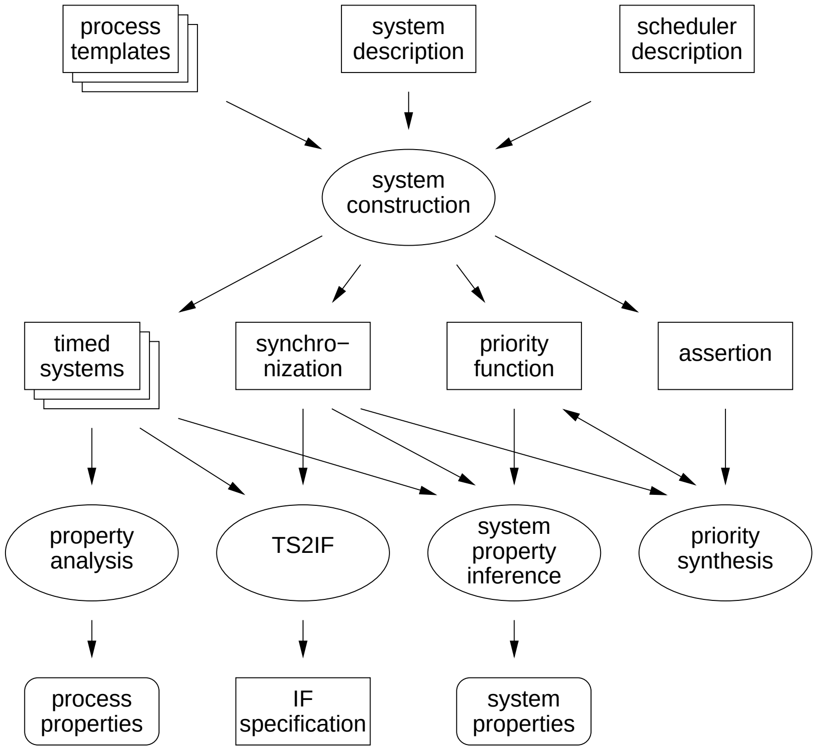


Figure 6.3: The internal structure of Prometheus.

of Prometheus, the processes specified in the system description file are read and internally represented as timed systems. For each process, a detailed diagnostic of its liveness properties is printed, consisting of the following data:

- Its (non)zenoness. If the process has zeno loops, one of them is printed. Otherwise, the minimal loop time (time between two occurrences of the same action) is printed.
- Its minimal inter-arrival time.
- Its local timelock-freedom. In the negative case, a list of all transitions that are not locally timelock-free, is printed.
- Its timelock-freedom.
- Its local livelock-freedom. In the negative case, a list of all states that are not locally livelock-free, is printed.
- Its livelock-freedom.

- If the process is not locally livelock-free, a diagnostic of the local deadlock-freedom is printed. In the negative case, a list of all actions that are not locally deadlock-free, is shown.
- Its structural liveness.

In case a process is not locally deadlock-free, PROMETHEUS tries in a backward fixpoint calculus to restrict the guards as in example 4.3 in order to make the process locally deadlock-free, and prints a new diagnostic of the modified process.

Next, PROMETHEUS introduces preemption transitions and places and resuming transitions for all states that use some preemptable resource, and synchronizes resource allocation actions according to Σ_{PMTN} with the compatible preemption actions. In contrast to the method proposed in section 3.2.2, the current version of PROMETHEUS allows no further preemption of a process that is preempted from one resource. According to the declaration of the resource using places of the processes, the priority function modelling mutual exclusion is automatically computed.

After the construction of the synchronized system of processes, the scheduler, if specified, is interpreted, and the priority function pr_{pol} representing the scheduling policy is added to the priority function computed so far.

Next, the transitive closure of the priorities is computed, and a warning message is issued if there are circuits in the priority relation. If the option **-u** is active, PROMETHEUS runs the priority synthesis algorithm 4.1 in order to make states where priorities form a circuit, as well as states where mutual exclusion on the resources, or the optional assertion is violated, unreachable, if this is not yet guaranteed by construction.

If the option **-d** is selected, then the system is made deterministic by completing the priority order associated with any system state to a total order on all controllable actions.

Hereafter, a diagnostic about safety and liveness of the composed system is printed, which has been derived from the individual diagnostics and the way the system is composed. In particular, PROMETHEUS indicates if the system is structurally safe, that is, whether "bad" states where mutual exclusion or the assertion are violated, or the priority function is not well-defined, are not reachable from the initial states of the system. If the hypotheses of the worst case completion time algorithm of section 5.6 are met, then the minimal inter-arrival time, the worst-case completion time, and their ratio are shown for each process. The minimal inter-arrival and maximal execution times are obtained automatically by analysing the timed systems modelling the processes.

If the option **-a** has been selected, the product timed system is constructed and written to a file. If one of the options **-i**, **-j** has been selected, the product system is output in the form of a timed automaton, or a timed automaton with convex invariants, respectively. The output of the product timed system allows, for small systems, to directly use KRONOS in order to carry out verification of the constructed system. States that are not reachable from the initial clock valuation where all clocks are zero, are eliminated if in addition option **-r** is chosen.

Without any of the options **-a**, **-i**, or **-j**, the set of processes, together with the synchronizations, is output as an IF specification, and the priority function is written to a separate file.

Execution 6.1 shows the output of PROMETHEUS for the crossover resource allocation example 4.8. Both processes are found to be structurally live. The minimal inter-arrival times of P_1 and P_2 are found to be 10 and 12, respectively. As we have already seen in example 3.6, the priority function $pr^{r_1}_{\mathsf{mutex}} \oplus pr^{r_2}_{\mathsf{mutex}}$ is not well-defined in state $(P_1.use_1, P_2.use_2)$, such that PROMETHEUS displays a warning message. The transitions $P_1.bgn_2$ and $P_2.bgn_1$ are found to be structurally unsafe. Since the option **-u** has been chosen, the priority synthesis algorithm is run and succeeds in making the system structurally safe, that is, making states where the priority function is not well-defined, or where mutual exclusion is violated, unreachable. As a product of structurally live systems, the whole system is structurally live. As we have seen in example 5.5, the worst-case completion time calculus finds that P_1 is blocked by P_2 during at most 3 time units. With an execution time of 6, the WCCT of P_1 is 9, and therefore lower than its minimal inter-arrival time of 10. Similarly, P_2 is blocked during at most 3 time units, and is guaranteed to complete before its deadline of 12. Finally, both the IF specification and the priority function are written to files. The whole execution takes about 0.2 seconds on a PentiumPro under Linux.

6.5 Discussion

Some modelling tools for real-time systems have already been mentioned in the introduction, such as GIOTTO [HHK01], PTOLEMY [L$^+$01], the SPI Workbench [EZR$^+$99], TAXYS [BPS00, CPP$^+$01], PARAGON [BALS99], and MetaH [Ves97]. All of these tools aim at quite different aspects of modelling, design, and verification.

Although PROMETHEUS is a modelling and design tool for real-time systems, too, its approach differs from the philosophy all of the tools cited above. Its main differences with respect to the latter are

- the compositional reasoning about safety, liveness, and timing properties, based on structural properties;
- verification of consistency between different properties; and
- the possibility of specifying a scheduler in a high-level modelling language.

The downside of this approach is that most of the compositional reasoning is conservative, and in case some property cannot be guaranteed by PROMETHEUS, help from "classical" verification techniques and tools is necessary. This help is provided by the validation platform IF, in which PROMETHEUS is integrated.

Execution 6.1 The resource allocation example 4.8.

```
rechasse> prometheus.i686 -u crossover.system
Properties of P1:
  structurally non-Zeno with minimal loop time 10
  minimal inter-arrival time 10
  locally timelock-free
  timelock-free
  locally livelock-free
  livelock-free
  structurally live

Properties of P2:
  structurally non-Zeno with minimal loop time 12
  minimal inter-arrival time 12
  locally timelock-free
  timelock-free
  locally livelock-free
  livelock-free
  structurally live

Computing mutex priorities.. done.
Checking consistency...
  Warning: priority circuit { P1.bgn2 P2.bgn2 } in (P1.use1 P2.use2 ) .
  done.
Checking for structural safety... done.
Structurally unsafe transitions:
    bgn1 from (P1.wait P2.use2 )
    bgn2 from (P1.use1 P2.wait )
Making bad states unreachable. done.

Product:
  structurally safe
  structurally live
  WCCT:
    P1: (P2, 3, 12)   WCCT=9  Tmin=10  Tmin/WCCT=1.111
    P2: (P1, 6, 10)   WCCT=9  Tmin=12  Tmin/WCCT=1.333

Writing IF file... done.
Writing priority file (2 entries).. done.
```

Chapter 7

Case Studies

7.1 An Active Structure Control System

We consider an embedded active structural control system, used to prevent damages on buildings caused by earthquakes or strong wind [ECB97, KLK00, Bra00]. It is composed of three processes. A cyclic process called *modeler* makes a computation which takes between 2.3 and 2.5 ms and updates a shared buffer, which takes 1 ms. A process of period between 3.2 and 14.5 ms called *pulser*, reads the data stored in the buffer and writes them to actuators. Reading takes 0.2 ms, writing 5.8 ms. Both *modeler* and *pulser* share the same processor. A further timing constraint to be respected requires that the data read by the pulser be fresh, that is, have been stored by the *modeler* at most 13 ms ago. Freshness is imposed by a process called *data*.

The three processes are modelled as in figure 7.1, where a time unit corresponds to 0.1 ms. It is easy to see that the processes *modeler* and *pulser* are structurally live. According to section 5.5, the scheduling requirement is $K_{\mathsf{sched}} = K_{\mathsf{sched}}^{modeler} \wedge K_{\mathsf{sched}}^{data} \wedge K_{\mathsf{sched}}^{pulser}$ with

$$\begin{aligned}
K_{\mathsf{sched}}^{modeler} &= sleeping_m \vee waiting_c \vee (computing \wedge x_m \leqslant 25) \vee \\
&\quad\; waiting_u \vee (updating \wedge x_m \leqslant 10) \\
K_{\mathsf{sched}}^{data} &= true \\
K_{\mathsf{sched}}^{pulser} &= (sleeping_p \wedge t_p \leqslant 145) \vee (waiting_r \wedge t_p \leqslant 85) \vee \\
&\quad\; (reading \wedge x_p \leqslant 2 \wedge t_p \leqslant 87 \wedge t_p - x_p \leqslant 85) \vee \\
&\quad\; (writing \wedge x_p \leqslant 58 \wedge t_p \leqslant 145 \wedge t_p - x_p \leqslant 87)
\end{aligned}$$

It is easy to see that $K_{\mathsf{sched}}^{modeler}$, $K_{\mathsf{sched}}^{data}$, and $K_{\mathsf{sched}}^{pulser}$ are inductive invariants of the processes *modeler*, *data*, and *pulser*, respectively.

The three processes share two resources — the processor CPU, and the *buffer* —

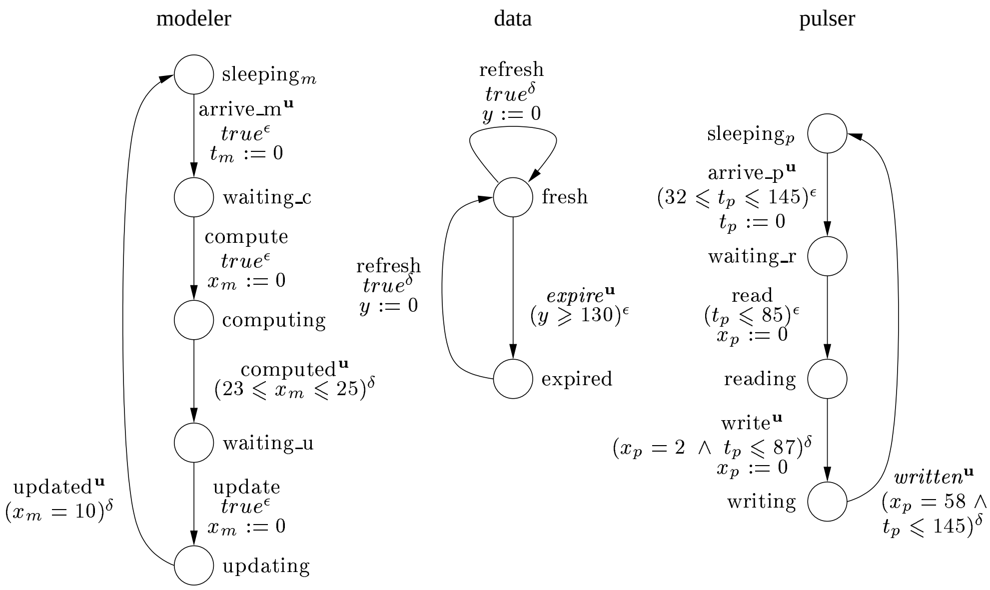


Figure 7.1: The active structural control system.

that are used in the following states:

$$
\begin{aligned}
U_{modeler,CPU} &= computing \vee updating \\
U_{modeler,buffer} &= updating \\
U_{pulser,CPU} &= reading \vee writing \\
U_{pulser,buffer} &= reading\ .
\end{aligned}
$$

In the composition of the three processes, the action *updated* synchronizes with *refresh*, which means that at the instant where the *modeler* finishes writing data to the buffer, *data* changes to (or remains in) the *fresh* state. Formally, $\Sigma = \{\ updated \mid refresh\}$. Maximal progress of the synchronization paradigm makes sure that *updated* never interleaves, since some transition labelled by *refresh* is always enabled. In order to prevent *refresh* from interleaving when the *modeler* is not ready to synchronize, we introduce the priority function

$$pr_{\mathsf{maxprg}} = \big\{\big(true,\ refresh \prec_\infty \{arrive\_m,\ compute,\ computed,\ update\}\big)\big\}\ .$$

Mutual exclusion for the two resources is ensured by

$$
\begin{aligned}
pr_{\mathsf{mutex}} &= pr^{CPU}_{\mathsf{mutex}} \oplus pr^{buffer}_{\mathsf{mutex}} \\
&= \Big\{\big((waiting\_c \ \vee\ waiting\_u) \ \wedge\ (reading \ \vee\ writing), \\
&\qquad \{compute,\ update\} \prec_\infty \{write,\ written\}\big), \\
&\quad \big((computing \ \vee\ updating) \ \wedge\ waiting\_r),\ read \prec_\infty \{computed,\ updated\}\big)\Big\} \ \oplus \\
&\quad \big\{\big(waiting\_u \ \wedge\ reading,\ update \prec_\infty write\big), \\
&\quad \big(updating \ \wedge\ waiting\_r,\ read \prec_\infty updated\big)\big\}
\end{aligned}
$$

We model the freshness constraint saying that *pulser* must not enter the state *reading* when *data* has expired, by the priority function

$$pr_{\mathsf{fresh}} = \{(\mathit{expired},\ \mathit{read} \prec_\infty \{\mathit{arrive\_m},\ \mathit{compute},\ \mathit{computed},\ \mathit{update},\ \mathit{updated}\})\}$$

ensuring that *pulser* blocks on *data_exp* if the state of *data* is *expired*, leaving priority to *modeler*.

Let

$$(TS, pr_{\mathsf{sync}}) = \left(\|_\Sigma \{\mathit{modeler}, \mathit{data},\ \mathit{pulser}\},\ pr_{\mathsf{maxprg}} \oplus pr_{\mathsf{mutex}} \oplus pr_{\mathsf{fresh}}\right)$$

denote the synchronized system. (TS, pr_{sync}) is not structurally safe with respect to the states verifying mutual exclusion, since there is no interleaving transition disabling transition *read* in the states *(updating, fresh, waiting_r)* and *(updating, expired, waiting_r)*. However, *read* is disabled by *updated*$\shortmid$*refresh*, according to pr_{mutex} and maximal progress: $\mathit{read} \prec_\infty \mathit{updated} \prec_\infty \mathit{updated}\shortmid\mathit{refresh}$. Therefore, (TS, pr_{sync}) is safe with respect to mutual exclusion.

The state constraint $K_{\mathsf{sched}} = K_{\mathsf{sched}}^{modeler} \oplus K_{\mathsf{sched}}^{data} \oplus K_{\mathsf{sched}}^{pulser}$ is not an inductive invariant of (TS, pr_{sync}). For example, the process *modeler* can be scheduled five times before giving the process *pulser* a turn. Hereafter, *pulser* will get deadlocked in *waiting_r*. Let us therefore add a scheduler to (TS, pr_{sync}), giving priority to *pulser* for access to the processor if *data* is *fresh*:

$$pr_{\mathsf{res}}^{CPU} = \{(\mathit{fresh},\ \{\mathit{compute} \prec_0 \mathit{read},\ \mathit{update} \prec_0 \mathit{read}\})\}\ .$$

K_{sched} is an inductive invariant of the scheduled system $(TS, pr_{\mathsf{sync}} \oplus pr_{\mathsf{res}}^{CPU})$, as can be verified for example using KRONOS. In fact, as soon as *pulser* blocks on expired *data*, its deadline is far enough for *modeler* to update the buffer before *pulser* misses its deadline. The priority function $pr_{\mathsf{sync}} \oplus pr_{\mathsf{res}}^{CPU}$ specifies synchronization between the processes, and the scheduler maintaining the scheduling requirement.

Let us now slightly modify this example. We strengthen the timing constraints and require a relative deadline of 200 ms for *modeler*. This is modelled by conjuncting the guard $x_m = 10$ of transition *updated* in figure 7.1 with the constraint $t_m \leqslant 200$, and strengthening accordingly the guards of the controllable actions of this new process *modeler'* in order to make it structurally live. The scheduling requirement is now

$$\begin{aligned} K_{\mathsf{sched}}^{modeler'} \quad = \quad & \mathit{sleeping}_m \ \vee\ (\mathit{waiting\_c} \ \wedge\ t_m \leqslant 165)\ \vee \\ & (\mathit{computing} \ \wedge\ x_m \leqslant 25 \ \wedge\ t_m \leqslant 190 \ \wedge\ t_m - x_m \leqslant 165)\ \vee \\ & (\mathit{waiting\_u} \ \wedge\ t_m \leqslant 190)\ \vee \\ & (\mathit{updating} \ \wedge\ t_m \leqslant 200 \ \wedge\ x_m \leqslant 10 \ \wedge\ t_m - x_m \leqslant 190)\ . \end{aligned}$$

The attempt to schedule the new synchronized system

$$(TS', pr_{\mathsf{sync}}) = \left(\|_\Sigma \{\mathit{modeler'}, \mathit{data},\ \mathit{pulser}\},\ pr_{\mathsf{maxprg}} \oplus pr_{\mathsf{mutex}} \oplus pr_{\mathsf{fresh}}\right)$$

as before by giving *pulser* priority over *modeler'* if *data* is *fresh*, fails. Timing analysis shows that $K'_{\mathsf{sched}} = K_{\mathsf{sched}}^{modeler'} \wedge K_{\mathsf{sched}}^{data} \wedge K_{\mathsf{sched}}^{pulser}$ is no more an inductive invariant of $(TS', pr_{\mathsf{sync}} \oplus pr_{\mathsf{res}}^{CPU})$, since the priority of *pulser* over *modeler'* as long as *data* is *fresh*, makes *modeler'* miss its deadline. Liveness of the whole system is reestablished by scheduling both processes under EDF if *data* is *fresh*. This policy is specified by the priority rule

$$\begin{aligned}(pr_{\mathsf{res}}^{CPU})' \quad = \quad & \big\{\big(\mathit{fresh} \wedge t_m - t_p < 200 - 145,\ \{\mathit{compute} \prec_0 \mathit{read},\ \mathit{update} \prec_0 \mathit{read}\}\big),\\ & \big(\mathit{fresh} \wedge t_m - t_p > 200 - 145,\ \{\mathit{read} \prec_0 \mathit{compute},\ \mathit{read} \prec_0 \mathit{update}\}\big)\big\}\ .\end{aligned}$$

Scheduler 7.1 shows the whole scheduler description. Analysis of the specification, and the output of the scheduled system by PROMETHEUS take less than a second.

Scheduler 7.1 EDF scheduler for the active structure application.

```
SCHEDULER edf_active_structure()
        NONPREEMPTABLE cpu
        PRIORITIES
                // prevent data from interleaving
                data.refresh << modeler.arrive_m
                data.refresh << modeler.compute
                data.refresh << modeler.computed
                data.refresh << modeler.update

                // block on expired model
                pulser << modeler if data.expired and pulser.waiting

                // conflict resolution using EDF
                modeler <0 pulser if [ modeler.t - pulser.t < 200-145 ]
                                     and data.fresh
                pulser <0 modeler if [ modeler.t - pulser.t > 200-145 ]
                                     and data.fresh
END edf_active_structure
```

Timing analysis of the scheduled system $\big(TS', pr_{\mathsf{sync}} \oplus (pr_{\mathsf{res}}^{CPU})'\big)$ using KRONOS shows that K'_{sched} is not an inductive invariant. However, the system is correctly scheduled from the initial state where both *modeler'* and *pulser* are *sleeping*, *data* is *expired*, and all clocks are zero. In other words, there is a non-empty inductive invariant of $\big(TS', pr_{\mathsf{sync}} \oplus (pr_{\mathsf{res}}^{CPU})'\big)$ implying K'_{sched} that holds for this initial state. We have found a feasible scheduler and obtained a correctly scheduled system by successively applying more and more "fine-grained" scheduling policies.

7.2 A Robotic Arm Controller

This case study is taken from [LV98]. It is about a robotic arm programmed to take objects from a conveyor belt, to store them in a buffer shelf, and to put them eventually into a basket. The arm is controlled by five tasks sharing one CPU:

- A *trajectory control* (TC) is spawned every 16ms to read commands from a shared buffer P_2 and issue set-points to the low-level arm controller. This task terminates immediately if there are no commands to process, otherwise it has an execution time between 5ms and 6ms. Its deadline is equal to the period of 16ms.

- Two motion executers (ME), a *lifter* and a *putter*, are invoked whenever the system must move an item; each ME generates a set of command in the command buffer. The *lifter* is activated whenever the system detects an object on the conveyor belt, with a minimum intertime of 40ms between each arrival. The *lifter* issues to the TC the commands to get the object and to put it in the buffer shelf P_1, and activates the *putter*, which sends to the TC the commands to take the object from the buffer shelf and put it into the basket. Both ME produce their commands within 4ms to 8ms.

- A *sensor reader* (SR) reads several sensors every 24ms. Its execution time is 1ms, and its deadline is equal to the period. The results of the SR are used by the TC.

- A *motion planner* (MP) refines the motion plan for the MEs each time it can be run without compromising the safeness of the remaining tasks. The MP tries to run once every 80ms and, if accepted, produces a refined plan within 10ms.

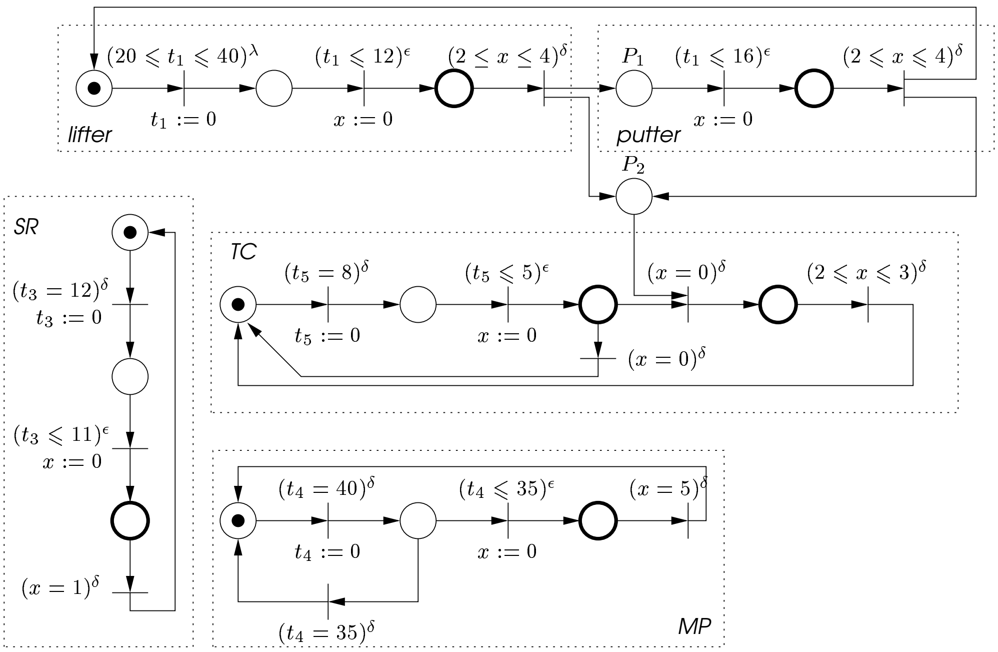


Figure 7.2: Petri net model of the robotic arm controller.

Figure 7.2 shows a model of the robotic arm controller as a Petri net to underscore the producer/consumer relation between *lifter*, *putter*, and *trajectory control*. Timing information is given in $2ms$ time units. In contrast to the original specification, we assume an upper bound of $80ms$ for the inter-arrival time of the sporadic process *lifter_putter*, in order to obtain a finite-state system that can be explored using IF. Places where the CPU is used, are drawn in bold.

Mutual exclusion on the CPU is ensured by a priority function $pr^{CPU}_{\mathsf{mutex}}$, similarly to the previous example. In order to ensure 1-safety of the Petri net, we bound the capacity of the buffer places P_1 and P_2 by one; for the sake of simplicity, the construction for the latter is omitted in the figure. Compared to [LV98], our model requires stronger synchronization and imposes harder schedulability constraints as the buffer size can not exceed one.

Notice that the worst case CPU utilization factor (the sum of the execution times divided by the minimal inter-arrival times) for the mandatory tasks (*trajectory control*, *lifter*, *putter*, and *sensor reader*) is 86%. This indicates that the scheduling constraints are really tight. We try rate-monotonic scheduling, generate the scheduled system using PROMETHEUS, explore it with IF, and are lucky: all processes are live and meet their timing constraints. Even the optional *motion planner* is scheduled in each period.

7.3 PATH's Automated Vehicle Control

PATH's ("Partners for Advanced Transit and Highways") Automated Vehicle Control and Safety System (AVCSS) project [Var93] encompasses controlling a set of cars moving autonomously in a *platoon* — that is, one car behind the other, with a small distance of about 4-6 meters — at high speed (about 100 km/h) on a highway equipped with magnets marking the center of a lane. The application, described in [Tri01, TY01a], is in charge of controlling the longitudinal and lateral position of each vehicle in the platoon. It consists of a set of concurrent processes running on the real-time operating system QNX, reading data from sensors (radar, speedometer, accelerometer, magnetometer, gyroscope), writing to actuators (throttle, brake, and steering), and communicating via a wireless network with other vehicles.

Figure 7.3 shows the hardware architecture of the application. Arrows stand for data flow between the different hardware units. Sensor data are sampled by analog/digital converters and read by a control computer, which also controls the different actuators, and communicates with the other vehicles. A human-machine interface (HMI computer) displays information to the passengers of the car. A laptop is used for initialisation.

Figure 7.4 shows the software architecture of the application. Solid boxes stand for hardware drivers, all other boxes for QNX processes. The processes communicate via a publish/subscribe middleware: as soon as some process writes a variable to a common database (which is not shown in the figure), all processes that have subscribed to that variable, are notified of its update, and can read variables from the database. Atomicity of the database operations serializes the requests, ensuring for example that the value read by a process corresponds to the last update.

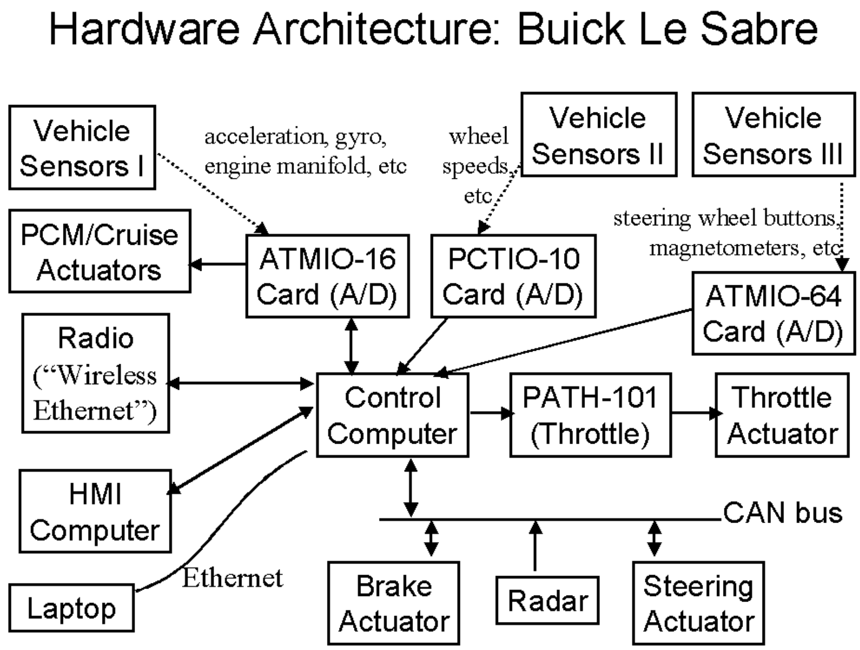


Figure 7.3: The AVCSS hardware architecture.

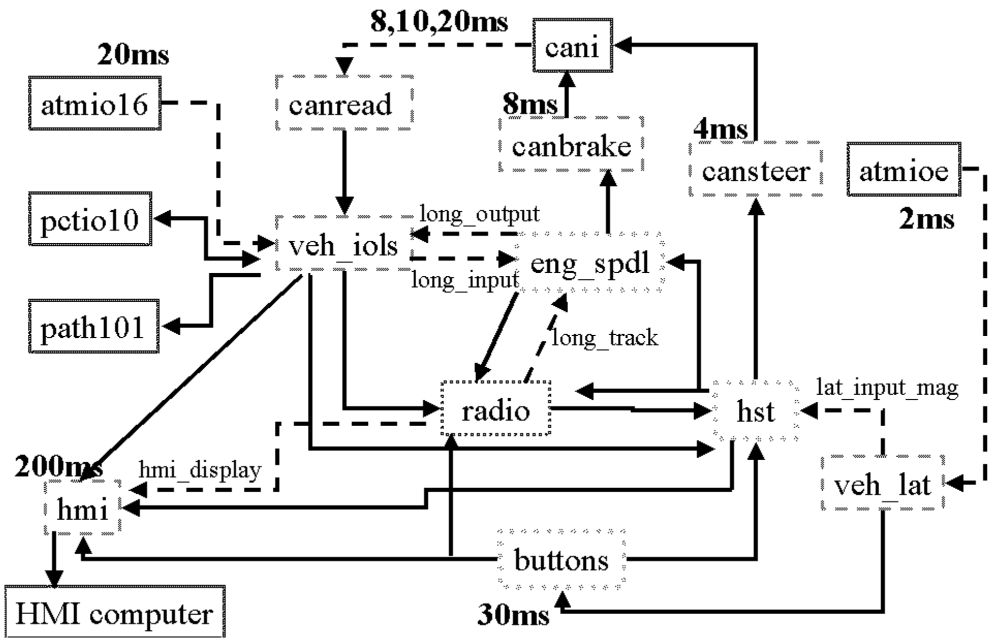


Figure 7.4: The AVCSS software architecture.

Processes can be event-triggered — which is the case for those having subscribed to database updates —, time-triggered (periodic), or both (the HMI process). Dashed arrows between processes represent events triggering the process they point to and are labelled by the name of the updated variable, dashed arrows from hardware units to processes represent periodic hardware interrupts triggering the process. Solid arrows between processes stand for dataflow via the database, solid arrows between a hardware unit and a process mean that the process synchronously reads data from, or writes data to, the corresponding hardware driver.

The processes *veh_iols*, *eng_spdl*, and *canbrake* are in charge of controlling the longitudinal position of the car in the platoon, the processes *veh_lat*, *hst*, and *cansteer* control its lateral position. *canread* is invoked with three different periods, corresponding to input from the three sources connected to the CAN bus. *radio* periodically communicates with the other vehicles, with a period of 20 ms. *hmi* and *buttons* are dedicated to interaction with the passengers.

7.3.1 Modelling the Platoon Application

We have chosen to model the application processes by 12 *logical* processes, gathering the physical QNX processes according to the control flow:

- an event-triggered process *database* modelling the update of variables triggering some process;
- a periodic process *lateral_input* consisting of the successive activation of *veh_lat* and *hst*;
- an event-triggered process *eng_spdl*;
- a periodic process *longitudinal* consisting of *veh_iols* reading data from the hardware drivers, triggering *eng_spdl* by updating the variable *long_input*, waiting for the update of *long_output*, and writing data to hardware drivers and the database;
- periodic processes *canbrake*, *cansteer*, *buttons*, and *radio*;
- a process *hmi* that is triggered both by the end of its period, and *radio*;
- three instances of the periodic process *canread*.

The processes have been modelled as discussed in chapter 3. The arrival transition of event-triggered processes synchronizes in a rendez-vous with the triggering transition. The deadline of the periodic processes is equal to their period, which means that there is one single instance of each logical process. The process models follow the description of the processes given in [Tri01, TY01a]; we only show the global system description (system 7.1), and the model of the process template *can_out* (process 7.1) as an example.

The only declared resource, the CPU, is non-preemptable. However, the execution of a process consists of a sequence of steps, with the CPU being freed at the end of

System 7.1 System description of the platoon application.

```
system path
        nonpreemptable cpu;

        // time unit: 40us

        // parameters: (priority, dummy period, t_async)
        database.proc           db (11, 0, 1)

        // parameters: (priority, period, t_read, t_update, t_irq,
        //              t_sync, t_switch)
        lateral_input.proc      li (10,  50, 1, 3, 1, 1, 1)
        eng_spdl.proc           es ( 5, 250, 1, 3, 1, 1, 1)
        longitudinal.proc       lo ( 1, 500, 1, 3, 1, 1, 1)
        radio.proc              ra ( 2, 500, 1, 3, 1, 1, 1)
        hmi.proc                hm ( 3, 500, 1, 3, 1, 1, 1)
        buttons.proc            bu ( 0, 750, 1, 3, 1, 1, 1)
        can_out.proc            cb ( 7, 200, 1, 3, 1, 1, 1),
                                cs ( 9, 100, 1, 3, 1, 1, 1)
        canread.proc            c1 ( 8, 200, 1, 3, 1, 1, 1),
                                c2 ( 6, 250, 1, 3, 1, 1, 1),
                                c3 ( 4, 500, 1, 3, 1, 1, 1)

        synchro
                lo.db_wrote_li!         AND db.li_updated!
                db.trigger_eng_spdl!    AND es.triggered_lat!
                es.db_wrote_lo!         AND db.lo_updated!
                db.trigger_veh_iols!    AND lo.db_triggered_lo!

                ra.db_wrote_lt!         AND db.lt_updated!
                db.trigger_eng_spdl2!   AND es.triggered_rad!

                ra.db_wrote_hmi!        AND db.hm_updated!
                db.trigger_hmi!         AND hm.triggered!

        scheduler
                rms.sched()
end path
```

each step, which makes the process execution non-atomic. Mutual exclusion on the other shared resources of the system — that is, the database, and the *cani* hardware driver — is ensured by mutual exclusion on the CPU, which is used in any state where some other resource is used. Timing information has been taken from [Tri01, TY01a]. One time unit corresponds to $40\mu s$, corresponding to an over-estimation of the time taken by the fastest middleware and QNX primitives relevant for the model.

The database is parameterized by its priority, a dummy period, and the time required for asynchronous message passing between processes. The parameters of the other processes are the priority, period, and the times for database read, database update, hardware interrupt handling, synchronous message passing between a process and a hardware driver, and context switching. Processes with the same periods are assigned different priorities; the actual process priorities are determined by the scheduler `rms.sched`, as discussed below.

The first group of synchronization declarations of system 7.1 models the process *longitudinal* updating the variable *long_input*, the database triggering *eng_spdl*, which updates *long_output*, causing the database to wake up the suspended *longitudinal* process. The second group of synchronizations stands for the *radio* process updating *long_track*, which triggers *eng_spdl*, and *radio* updating the variables *maneuver_des* and *hmi_display*, triggering the process *hmi*. All synchronizations are blocking rendez-vous.

The processes *canbrake* and *cansteer* (process 7.1) acquire the CPU to read one variable (*long_output* and *lat_output*, respectively), free the CPU, and request it a second time to write their output to the CAN bus driver *cani*, before completing their execution. Context switching times are taken into account at the first operation of each process, and on invocation of other pieces of code (the CAN driver, in this case).

The whole model consists of 12 processes with up to 13 control states. The product timed system would have 3×10^9 control states, which motivates the interest of applying scheduling policies to restrict the possible behaviours of the system, and reduce its state space.

7.3.2 Analysis and Scheduler Design

PROMETHEUS reports the synchronized system to be structurally safe with respect to mutual exclusion on the CPU. Due to blocking rendez-vous modelling variable updates and triggering of processes, the system is not structurally live, but it is found to be structurally non-zeno and timelock-free. The knowledge about timelock-freedom of the system is essential for proving liveness — that is, timelock-freedom and livelock-freedom — of its processes. If timelock-freedom can be shown by PROMETHEUS applying the compositionality results of chapter 4, then only livelock-freedom of the processes needs to be checked using more expensive algorithms and tools such as KRONOS or IF.

According to the system specification, the database runs with the highest priority, which means in particular that database operations are atomic. In order to reduce the state space of the transition system generated by IF, we have assigned arbitrary priorities also between processes of the same period, making conflict resolution deterministic. Scheduler 7.2 shows the scheduler description file.

The whole analysis and generation of the output files by PROMETHEUS take about three seconds. Exploring the state-space of the scheduled system using IF generates a

Process 7.1 The *can_out* process template used to instantiate *canbrake* and *cansteer*.

```
pnd can_out (prio, period, t_read, t_update, t_irq, t_sync, t_switch)
        places sleeping w_db u_db w_cani u_cani
        resource cpu: u_db u_cani
        clocks t x
        init: sleeping;

        arrive:         UNCTRL
                        in: sleeping
                        out: w_db
                        guard t=$period delayable
                        resets t;

        db_read:        in: w_db
                        out: u_db
                        guard true eager
                        resets x;

        db_rdone:       UNCTRL
                        in: u_db
                        out: w_cani
                        guard x=$t_switch+$t_read delayable;

        hw_write:       in: w_cani
                        out: u_cani
                        guard true eager
                        resets x;

        hw_wdone:       UNCTRL
                        in: u_cani
                        out: sleeping
                        guard x=$t_sync+$t_switch delayable;
end can_out
```

transition system with about 180000 states. All processes meet their timing constraints and are live in the scheduled system.

A Two-Processor Specification

The original platoon application runs on a monoprocessor platform for each car. However, it would be interesting to partition the application processes on two or more CPUs according to functional or reliability aspects, or processor utilization. We suppose that the processes *lateral_input* and *cansteer*, which are in charge of the lateral

Scheduler 7.2 RMS scheduler for the platoon application.

```
scheduler rms()
    priorities
        forall Pi: process
        forall Pj: process
            Pi <0 Pj if $Pj.period < $Pi.period or
                        $Pj.period = $Pi.period and $Pj.prio > $Pi.prio
end rms
```

control, will run on one CPU, and all other processes run on a second CPU. Both CPUs are assumed to be half as fast as previously. The CPU loads are 88% and 55% respectively. Since database operations are twice as long as before, the database load is now 93%, which makes scheduling a hard problem.

In addition to the CPUs, mutual exclusion on the shared resources *database* and *cani* must be explicitly ensured now, since their use is not atomic any more. This means that the model does not fit into classical rate-monotonic scheduling theory any more. [Bal98] proposes a fixed-priority assignment algorithm for partially ordered systems of tasks reacting to external events or other tasks. However, this algorithm does not apply to multi-processor scheduling neither, such that feasibility of a given scheduler must be determined by other means.

We take our modified scenario to illustrate the design process proposed in section 5.7. Analysis of the synchronized system has already been discussed in the previous section, so that we focus on the design of a feasible scheduler here.

We start with the same rate-monotonic scheduler 7.2 as before, which resolves conflicts between processes contending for the same resource according to their periods. A look at the transition system generated by IF shows that process *c3* can acquire CPU_1 and the database while process *lateral_input* is on CPU_2. When *lateral_input* tries to acquire the database, it blocks and misses its deadline before *c3* frees the database. That is, rate-monotonic scheduling is not feasible.

There are several conceivable remedies for this problem, among them

- applying a "better" scheduling policy such as EDF. However, the system scheduled under EDF turns out to suffer from the same problem;
- use a variant of the priority ceiling protocol to minimize blocking times. Unfortunately, the priority ceiling protocol is too pessimistic and does not allow using the resources in an efficient way (for example, access to the CAN driver is refused to any process as long as some other process is using the database). Due to the high database utilization, the priority ceiling protocol is not feasible;
- use of clairvoyant scheduling: when *lateral_input* will soon request the database, access to the latter is refused to any lower-priority processes;
- if no feasible scheduler can be found, using more powerful CPUs.

We try the third point, clairvoyant scheduling, by adding to the scheduler an admission control policy refusing lower-priority processes access to the database when *lateral_input* is in the control state preceding its request of the database. However, this admission control rule means that processes other than *lateral_input* are prevented from acquiring access to the database not only while *lateral_input* is using it, but already before, that is, during 67% of the time. Since the database utilization of the other processes is 31%, this causes *eng_spdl*, *longitudinal*, *radio*, and *hmi* to miss their deadlines when waiting for the database.

Therefore, it seems necessary to increase CPU power by, say, 20%, reducing database utilization to 77%. Under rate-monotonic scheduling, *eng_spdl* turns out to miss its deadline, which causes *longitudinal* and *radio* to deadlock.

In addition to RMS priorities, we now assign priorities between processes of the same period according to their database utilization using scheduler 7.2, motivated by the fact that the database is the most critical resource. It takes PROMETHEUS about one minute to analyse the specification, build the priority function, and output the scheduled system. Exploration of the latter by IF terminates after exploring half a million states. All processes meet their timing constraints, which means that scheduler 7.2 is feasible.

7.4 Discussion

The work on the case studies has confirmed the applicability of our approach, and usability of PROMETHEUS in practice. The approach seems especially interesting for modelling and analysing systems with many shared resources, complex inter-process coordination, and that do not fit in results from scheduling theory, or whose resource utilization is higher than allowed by the schedulability criterion.

Since the priority function modelling inter-process coordination grows with the complexity of the scheduler, the scheduler specification should be as detailed as necessary to be feasible, but as simple as possible. The reuse of scheduler specifications (in particular, RMS, PCP, and variants of them) saves time, and avoids specification errors. The consistency check based on priority functions helped indeed to discover some errors in the scheduler specification.

The ease of using PROMETHEUS to model processes, their synchronization, and the scheduler in a high-level formalism, has turned out to be essential to efficiently design and modify the models in our case studies. As a side effect, many low-level modelling errors, which can be hard to find, are avoided.

The platoon application has already been analysed in [Tri01, TY01a, TY01b]. However, in both cases, the approach differs from ours: [Tri01] models the applications so as to fit it into the framework of rate-monotonic analysis, which does not apply to arbitrary models (such as the modified scenario above). In [TY01a], verification of the timing constraints by the model is checked using on-the-fly state-space exploration without constraining the behaviour of the synchronized system by a particular scheduling policy. Moreover, the TAXYS tool suite [BPS00, CPP+01] used in [TY01a] allows automatic code generation. This approach is very efficient, but in case of non-respect of timing constraints, the whole application model has to be re-engineered without the possibility of excluding undesired behaviour by specifying a suited scheduling policy. It is therefore interesting and planned to integrate the possibility of specifying scheduling policies in TAXYS.

As the three applications were treated successfully with PROMETHEUS, the next step will be the modelling and analysis of more complex examples, using an improved version of PROMETHEUS.

Chapter 8

Conclusion

We have presented a general modelling framework for reactive real-time systems, based on timed systems with priorities. Both discrete and continuous-time semantics are supported. The novelty of our approach consists in the integration of scheduling policies with a description of the processes and their coordination, in a compositional framework.

Real-time systems are structured in a layered architecture, consisting of the application processes, a synchronization layer ensuring functional aspects of process coordination such as mutual exclusion, preemption, or atomicity, and a scheduler granting access to shared resources by applying conflict resolution and admission control policies.

We have proposed guidelines for modelling processes and their synchronization. Several frequently used scheduling policies, among them rate-monotonic and earliest-deadline-first scheduling, and the priority ceiling protocol, have been modelled as priority functions. Modelling separately admission control and conflict resolution, allows modular and predictable scheduler specifications that can easily be refined.

Priority functions are at the same time a powerful and, as we believe, a natural way of modelling coordination between processes. We have argued in favour of representing both functional and non-functional properties of inter-process coordination in a modular way as priority functions. By composition of the latter we get a priority function modelling overall coordination. This incremental modelling principle allows to verify consistency of the different behavioural requirements, and provides diagnostics in case of inconsistency, helping to detect specification errors at an early stage in the design process. Consistency of the priority functions and common-sense structural properties of the process models, which are sufficient conditions for safety and liveness, support the design of robust and predictable systems.

Results based on structural properties of processes allow compositional modelling and reasoning about liveness, safety, timing, and in certain cases correctness by construction. They are efficient since they do not require constructing the product system. We have also shown how it is possible to take advantage of the priority function modelling coordination for timing analysis of quite general models of processes and their interaction.

Based on these results, we have proposed an incremental design process for reactive real-time systems.

We have developed a tool called PROMETHEUS implementing the theoretical results, in particular, modelling based on priority functions, and compositional reasoning about properties of the scheduled system. Processes, synchronization, and the scheduler are specified in a high-level modelling language. Process and scheduler templates can be defined to build a library of process and scheduler types. PROMETHEUS is embedded in the platform IF, which allows to export PROMETHEUS models towards a large range of existing validation tools.

Models of real-world systems have been treated successfully with PROMETHEUS; the results are conclusive. We have designed feasible schedulers for real-time systems in absence of applicable results from scheduling theory. The expressive power and the high-level modelling formalism of PROMETHEUS have turned out to be essential for efficiently designing and engineering the models of real-time applications, and in particular schedulers, in our case studies. As a feedback from experimentation, experiences with modelling and analysing real-time applications have inspired several theoretical results, algorithms, and features of PROMETHEUS.

Perspectives

Currently much effort is made to integrate schedulability analysis of real-time systems, and code generation [Ves97, HHK01, BPS00, CPP$^+$01]. This is particularly interesting because it allows to obtain an implementation corresponding to the verified model, and rules out errors introduced by manual coding. At this end, the code of a process is annotated with timing constraints resulting from execution on a target machine. The difficulty consists in constructing, from the timed model of processes, their scheduling requirements, and a specification of the environment, a feasible scheduler.

In the TAXYS project [BPS00, CPP$^+$01], the system and its environment are specified in ESTEREL annotated with timing constraints, which can be verified, and compiled to C code. It is planned to allow the specification of scheduling policies and to generate a scheduler at compilation time, using results of this thesis. Similarly, a new project called EXPRESSO aims at the schedulability analysis, and scheduler generation, for annotated Real-Time JAVA [Gro00] code. We plan to use PROMETHEUS to generate an intermediate timed model for timing analysis, and scheduler construction. The scheduler could be generated by exploiting information provided by the priority function specifying overall coordination of the processes.

In both cases, the problem amounts to extending a compiler for untimed systems, to a compiler that supports the additional dimension *time*, generating functionally correct programs whose correct timing is equally guaranteed. This research line could lead to object-oriented real-time programming where objects declare their timing constraints, taken into account by the compiler for the construction of a feasible scheduler.

Another interesting work direction at shorter term seems to be an improvement of the algorithm computing a worst-case completion time based on priority functions, so as to allow an exact characterization for schedulability for a certain class of models, rather than a conservative estimation. An exact characterization would make the use of more expensive algorithms obsolete in determining feasibility of the scheduled system.

In the same perspective of strengthening compositional reasoning, the definition, for each timed system, of an interface with input and output events and timing assertions, would provide additional information allowing less conservative results for preservation of structural liveness.

Priority functions can be seen as types of behaviour. Taking up ideas in [Lee00], it seems an interesting perspective at longer term, to use priority functions, or a variant of them, to type the allowed behaviour of objects in object-oriented real-time languages. An interface declaration carrying this type would provide not only static information, but also information about the dynamical behaviour of the object, for example, in which order its methods can be called. Type-theoretical results could then be applied to check behavioural consistency of cooperating objects.

Bibliography

[AAE98] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *ACM Symposium on the Principles of Distributed Programming (PODC)*, 1998.

[ABR91] N.C. Audsley, A. Burns, and M.F. Richardson. Hard real-time scheduling: the deadline monotonic approach. In *Workshop on real-time operating systems and software*, 1991.

[ACD98] G. S. Avrunin, J. C. Corbett, and L. K. Dillon. Analyzing partially-implemented real-time systems. *IEEE Transactions on Software Engineering*, 24(8), 1998.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, , and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AGP+99] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. RTSS 1999*, pages 154–163. IEEE Computer Society Press, 1999.

[AGS00] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In M. Joseph, editor, *proc. FTRTFT 2000*, volume 1926 of *LNCS*, pages 106–120. Springer-Verlag, 2000.

[AGS01] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems, special issue on "control-theoretical approaches to real-time computing" (to appear)*, 2001.

[Alt98] K. Altisen. Génération automatique d'ordonnancements des systèmes temporisés. Master's thesis, ENSIMAG, INPG, Grenoble, France, 1998. In French.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Proc. Hybrid Systems II*, volume 999 of *LNCS*. Springer-Verlag, 1995.

[BACC+98] H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. Si 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.

[Bal98] F. Balarin. Priority assignment for embedded reactive real-time systems. In *Proc. ACM SIGPLAN Workshop LCTES'98*, volume 1474 of *LNCS*. Springer-Verlag, 1998.

[BALS99] H. Ben-Abdallah, I. Lee, and O. Sokolsky. Specification and analysis of real-time systems with PARAGON. In *Annals of Software Engineering*, 1999.

[BBK86] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, IX(2):127–168, 1986.

[BCL99] G. Bhat, R. Cleaveland, and G. Lüttgen. A practical approach to implementing real-time semantics. *ASE*, 7:127–155, 1999.

[Bel90] F. Belik. An efficient deadlock avoidance technique. *IEEE Trans. on Computers*, 39(7):882–888, 1990.

[Ber93] G. Berry. Preemption in concurrent systems. In *Proc. FSTTCS 93*, volume 761 of *LNCS*, pages 72–93. Springer-Verlag, 1993.

[Bes94] A. Bestavros. An ounce of prevention is worth a pound of cure. In *Proc. 9th Annual IEEE Conference on Computer Assurance (COMPASS'94)*, 1994.

[BFG+00] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: A validation environment for timed asynchronous systems. In E. A. Emerson and A. P. Sistla, editors, *Proc. CAV'00*, volume 1855 of *LNCS*, pages 543–547. Springer-Verlag, 2000.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BG98] G. P. Brat and V. K. Garg. Analyzing non-deterministic real-time systems with (max,+) algebra. In *Proc. RTSS'98*, pages 210–219, 1998.

[BGL97] P. Brémond-Grégoire and I. Lee. A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189, 1997.

[BGS00] S. Bornot, G. Gößler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS 2000*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.

[Bin97] P. Binns. Incremental rate monotonic scheduling for improved control system performance. In *Proc. RTSS'97*, pages 80–90, 1997.

[BLMSV98] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. Scheduling for embedded real-time systems. *IEEE Design & Testing of Computers*, 15(1):71–82, 1998.

[BMT99] M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In L. Pierre and T. Kropf, editors, *Proc. CHARME'99*, volume 1703 of *LNCS*, pages 125–141. Springer-Verlag, 1999.

[Bor98] S. Bornot. *De la Composition des Systèmes Temporisés.* PhD thesis, Université Joseph Fourier, Grenoble, France, 1998. In French.

[BPS00] V. Bertin, M. Poize, and J. Sifakis. Towards validated real-time software. In *Proc. 12th Euromicro Conference on Real Time Systems*, pages 157–164, 2000.

[Bra00] V. A. Braberman. *Modeling and Checking Real-Time System Designs.* PhD thesis, Departamento de Computación, Universidad de Buenos Aires, Buenos Aires, Argentina, 2000.

[BS97] S. Bornot and J. Sifakis. On the composition of hybrid systems. In *International NATO School on "Verification of Digital and Hybrid Systems"*, LNCS, pages 347–359. Springer-Verlag, 1997.

[BS98] S. Bornot and J. Sifakis. On the composition of hybrid systems. In *Proc. International Workshop "Hybrid Systems: Computation and Control"*, LNCS, pages 49–63. Springer-Verlag, 1998.

[BS00] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163:172–202, 2000.

[BSS97] A. Bouajjani, S.Tripakis, and S.Yovine. On-the-fly symbolic model-checking for real-time systems. In *Proc. IEEE Real-Time Systems Symposium, RTSS'97.* IEEE Computer Society Press, 1997.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Proc. COMPOS'97*, volume 1536 of *LNCS*. Springer-Verlag, 1998.

[BV95] G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Transactions on Software Engineering*, 21(12), 1995.

[BW95] Alan Burns and Andy Wellings. *HRT-HOOD — A Structured Design Method for Hard Real-Time Ada Systems.* Elsevier, 1995.

[CCMM95] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *Proc. Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Ordonnancement temps réel*. Hermes, 2000. In French.

[CH90] R. Cleaveland and M. Hennessy. Priorities in process algebras. *Information and Computation*, 87(1/2):58–77, 1990.

[CLN98] R. Cleaveland, R. Lüttgen, and V. Natarajan. A process algebra with distributed priorities. *Theoretical Computer Science*, 195(2):227–258, 1998.

[CLN01] R. Cleaveland, G. Lüttgen, and V. Natarajan. *Priority in Process Algebra*, chapter 12 of Handbook of Process Algebra, pages 711–765. North-Holland, 2001.

[CMSV99] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and D. Peled, editors, *Proc. CAV'99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

[Cor96a] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), 1996.

[Cor96b] J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Transactions on Software Engineering*, 22(7), 1996.

[CPP$^+$01] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS = ESTEREL + KRONOS. a tool for the development and verification of real-time embedded systems. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 391–395. Springer-Verlag, 2001.

[CT94] C.-M. Chen and S. K. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, University of Maryland, 1994.

[CW95] J. Camilleri and G. Winskel. CCS with priority choice. *Information and Computation*, 116(1):26–37, 1995.

[DB98] B. Dobbing and A. Burns. The ravenscar tasking profile for high integrity real-time programs. In *Proc. ACM SigAda Annual Conference*, pages 1 – 6. ACM Press, 1998.

[dB99] L. du Bousquet. Feature interaction detection using testing and model-checking. In *Proc. Formal Methods '99*, volume 1708 of *LNCS*, pages 622–641. Springer-Verlag, 1999.

[Die99] H. Dierks. Synthesizing controllers from real-time specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):33–43, 1999.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[DS00] B. Dutertre and V. Stavridou. Formal analysis for realtime scheduling. In *19th AIAA/IEEE Digital Avionics Systems Conference*, 2000.

[Dut00] B. Dutertre. Formal analysis of the priority ceiling protocol. In *IEEE Real-Time Systems Symposium (RTSS'00)*, pages 151–160, 2000.

[ECB97] W. Elseaidy, R. Cleaveland, and J. Baugh Jr. Modeling and verifying active structural control systems. *Science of Computer Programming*, 29(1-2):99–122, 1997.

[EWY99] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proc. RTCSA '99*. IEEE Computer Society Press, 1999.

[EZR+99] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich. Hardware/software codesign of embedded systems — the SPI workbench. In *Proc. Int. Workshop on VLSI, Orlando, Florida*, 1999.

[FC93] A. N. Fredette and R. Cleaveland. RTSL: A language for real-time schedulability analysis. In *Proc. RTSS'93*, pages 274–283. Computer Society Press, 1993.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. CAV '96*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, 1996.

[FJJV97] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.

[FKU95] C. Fidge, O. Kearney, and M. Utting. Interactively verifying a simple real-time scheduler. In P. Wolper, editor, *Proc. CAV'95*, volume 939 of *LNCS*. Springer-Verlag, 1995.

[Gar00] J. M. Garrido. *Performance Modeling of Operating Systems Using Object-Oriented Simulation*. Kluwer Academic / Plenum Publishers, 2000.

[GMR95] L. George, P. Muhlethaler, and N. Rivierre. Optimality and non-preemptive real-time scheduling revisited. Technical Report 2516, INRIA, 1995.

[Gößl98] G. Gößler. Modélisation et contrôle des systèmes multimédia. Master's thesis, ENSIMAG, Grenoble, France, 1998. In French.

[Gößl01] G. Gößler. PROMETHEUS — a compositional modeling tool for real-time systems. In P. Pettersson and S. Yovine, editors, *Proc. Workshop RT-TOOLS 2001*. Technical report 2001-014, Uppsala University, Department of Information Technology, 2001.

[Gro00] Real-Time Java Working Group. International j consortium specification. Technical report, International J Consortium, 2000.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report 2966, INRIA, 1996.

[HHK96] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In R. Alur and T. A. Henzinger, editors, *Proc. CAV'96*, volume 1102 of *LNCS*, pages 423–427. Springer-Verlag, 1996.

[HHK01] T. A. Henzinger, B. Horowitz, and C. Meyer Kirsch. Embedded control systems development with Giotto. In *Proc. LCTES 2001 (to appear)*, 2001.

[HHWT97] T.A. Henzinger, P.H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. In *Software Tools for Technology Transfer*, pages 110–122, 1997.

[HKL94] M. G. Härbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, 1994.

[HKO+93] M.G. Härbour, M.H. Klein, R. Obenza, B. Pollak, and T. Ralya. *A Practitioner's Handbook for Real-Time Analysis.* Kluwer, 1993.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proc. 7th Symp. on Logics in Computer Science (LICS'92) and Information and Computation*, volume 111, pages 193–244, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HR95] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117:221–239, 1995.

[IMA99] M. V. Iordache, J. O. Moody, and P. J. Antsaklis. A method for deadlock prevention in discrete event systems using Petri nets. Technical report, University of Notre Dame, 1999.

[IT99] ITU-T. Recommendation Z.100. Specification and Design Language (SDL). Technical Report Z-100, International Telecommunication Union — Standardization Sector, Geneva, 1999.

[JJ01] T. Janowski and M. Joseph. Dynamic scheduling and fault-tolerance: Specification and verification. *Int. Journal of Time-Critical Computing Systems*, 20:51–81, 2001.

[JLS00] H. Jensen, K.G. Larsen, and A. Skou. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In *Proc. FTRTFT 2000*, LNCS. Springer-Verlag, 2000.

[Kes97] S. Keshav. *An Engineering Approach to Computer Networking*. Addison-Wesley, 1997.

[KLK00] I. Kang, I. Lee, and Y. S. Kim. An efficient space generation for the analysis of real-time systems. *IEEE Transactions of Software Engineering*, 26(5), 2000.

[KLP+98] H.-H. Kwak, I. Lee, A. Philippou, J.-Y. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *Proc. RTSS 1998*, pages 409–418. IEEE Computer Society Press, 1998.

[KPSY92] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Integration graphs: A class of decidable hybrid systems. *Information and Computation*, 736, 1992.

[L+01] E. A. Lee et al. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, University of California at Berkeley, 2001.

[LA99] K. Lundqvist and L. Asplund. A formal model of a run-time kernel for Ravenscar. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications — RTCSA'99*, pages 504–507, 1999.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LAM99] K. Lundqvist, L. Asplund, and S. Michell. A formal model of the Ada Ravenscar tasking profile; protected objects. In *Proc. Ada-Europe '99*, LNCS. Springer-Verlag, 1999.

[LBGG94] 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, Special Issue on Real-Time Systems*, 1 1994.

[Lee98] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering, special volume on real-time software engineering*, 1998.

[Lee00] E. A. Lee. What's ahead for embedded software? *IEEE Computer*, pages 18–26, 9 2000.

[LJ99] Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Transactions on Programming Languages and Systems*, 21(1):46–89, 1999.

[LJ01] Z. Liu and M. Joseph. Verification, refinement and scheduling of real-time programs. *TCS*, 253:119–152, 2001.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm — exact characterization and average case behavior. In *Proc. RTSS 1989*, pages 166–171. IEEE, 1989.

[LV98] M. Lusini and E. Vicario. Static analysis and dynamic steering of time-dependent systems using Petri nets. Technical Report 28.98, University of Florence, 1998.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E.W. Mayr and C. Puech, editors, *STACS'95*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag, 1995.

[MTdR96] A. K. Mok, D.-C. Tsou, and R. C. M. de Rooij. The MSP.RTL real-time scheduler synthesis tool. *IEEE*, pages 118–128, 1996.

[Nak01] C. Nakhli. Spécification compositionelle à l'aide de priorités dynamiques. Master's thesis, Ecole Nationale d'Ingénieurs de Tunis, Tunisia, 2001. In French.

[NY00] P. Niebert and S. Yovine. Computing optimal operation schemes for chemical plants in multi-batch mode. In *Proceedings of Hybrid Systems, Computation and Control*, volume 1790 of *LNCS*. Springer-Verlag, 2000.

[Pet99] P. Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Uppsala University, 1999.

[Phi01] I. Phillips. CCS with priority guards. In K. G. Larsen and M. Nielsen, editors, *Proc. CONCUR 2001*, volume 2154 of *LNCS*, pages 305–320. Springer-Verlag, 2001.

[Ree97] G. Reeves. What really happened on mars? posted to Comp.risks, appeared in RISKS-19.49, 1997.

[RSLR95] R. Rajkumar, L. Sha, J. P. Lehoczky, and K. Ramamritham. *An Optimal Priority Inheritance Policy For Synchronization in Real-Time Systems*, chapter 11, pages 246–268. Prentice Hall, 1995.

[Rus99] J. Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. Technical Report CR-1999-209347, NASA Langley Research Center, 1999.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1), 1987.

[SK00] M. Saksena and P. Karvelas. Designing for schedulability: Integrating schedulability analysis with object-oriented design. In *Proc. 12th Euromicro Conference on Real-Time Systems*, 2000.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SSNB95] J. A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

[TA99] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proc. Formal Methods '99*, volume 1708 of *LNCS*, pages 233–252. Springer-Verlag, 1999.

[Tri98] S. Tripakis. *L'Analyse Formelle des Systèmes Temporisés en Pratique.* PhD thesis, Université Joseph Fourier, Grenoble, France, 1998.

[Tri99] S. Tripakis. Verifying progress in timed systems. In *Proc. ARTS'99*, LNCS. Springer-Verlag, 1999.

[Tri01] S. Tripakis. Embedded software architecture of an automated vehicle control system. Technical Report UCB/ERL M01/6, University of California, Berkeley, 2001.

[TY01a] S. Tripakis and S. Yovine. Modeling, timing analysis and code generation of PATH's automated vehicle control application software. In *Proc. CDC'01 (to appear)*, 2001.

[TY01b] S. Tripakis and S. Yovine. Timing analysis and code generation of vehicle control software using taxys. In *Proc. Workshop on Runtime Verification, RV'01*, volume 55. Elsevier, 2001.

[Var93] P. Varaiya. Smart cars on smart roads: Problems of control. *IEEE Transactions on Automatic Control*, 38(2):195–207, 2 1993.

[Ves94] S. Vestal. Fixed-priority sensitivity analysis for linear compute time models. *IEEE Transactions on Software Engineering*, 20(4):308–317, 1994.

[Ves97] S. Vestal. MetaH support for real-time multi-processor avionics. In *Proc. IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 11–21, 1997.

[Ves99] S. Vestal. Modeling and verification of real-time software using extended linear hybrid automata. Technical report, Honeywell Technology Center, 1999.

[Ves00] S. Vestal. Modeling and verification of real-time software using extended linear hybrid automata. In *Fifth NASA Langley Formal Methods Workshop*, 2000.

[WBC+00] D. Weil, V. Bertin, E. Closse, M. Poize, P. Vernier, and J. Pulou. Efficient compilation of esterel for real-time embedded systems. In *Proc. CASES'2000*, 2000.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.

[XP93] J. Xu and D. L. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, 1993.

[XP00] J. Xu and D. L. Parnas. Priority scheduling versus pre-run-time scheduling. *Time-Critical Computing Systems*, 18(1):7–23, 2000.

[Yov97] S. Yovine. KRONOS: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, 1997.

Appendix A

Proofs

Proof of proposition 2.1

For state $(s, \mathbf{x}) \in S \times \mathbb{D}^m$, let $\prec = pr(s, \mathbf{x})$ be the priority order associated with $(s, \mathbf{x})$ by pr. If $T(s) = \emptyset$, the equation

$$\lozenge^{\mathbf{b}_s} \bigvee_{t_i \in T(s)} g'_i \quad = \quad \lozenge^{\mathbf{b}_s} \bigvee_{t_i \in T(s)} g_i$$

trivially holds. Suppose that $T(s) \neq \emptyset$. We proceed by structural induction.

- Let A_0 be the set of actions leaving s that are maximal in $\prec$. $A_0 \neq \emptyset$, since $\prec$ is a priority order. For any action $a_i \in A_0$, a_i will, by backward closure of the C^j, remain maximal in all priority orders $\prec^j$ associated with $(s, \mathbf{x} + t\mathbf{b}_s)$ for all $t \geqslant 0$. Thus, $(\lozenge g'_i)(\mathbf{x}) = (\lozenge g_i)(\mathbf{x})$.

- For $n \geqslant 0$, suppose that $(\lozenge^{\mathbf{b}_s} \bigvee_{a_i \in A_n} g'_i)(\mathbf{x}) = (\lozenge^{\mathbf{b}_s} \bigvee_{a_i \in A_n} g_i)(\mathbf{x})$ holds, and let A_{n+1} be the set of the actions that are not dominated in $\prec$ by actions not in A_n (thus, $A_n \subseteq A_{n+1}$). For any $a_i \in A_{n+1} \setminus A_n$, we have by definition 2.8,

  $$(g'_i \ \vee \bigvee_{a_j \in A_n} \lozenge_k^{\mathbf{b}_s} g'_j)(\mathbf{x}) \quad = \quad (g_i \wedge \neg(\bigvee_{a_j \in A_n} \lozenge_k^{\mathbf{b}_s} g_j) \vee \bigvee_{a_j \in A_n} \lozenge_k^{\mathbf{b}_s} g'_j)(\mathbf{x}) \; .$$

  By distributivity of $\lozenge$ over the disjunction and additivity of $\lozenge$,

  $$\Big(\lozenge^{\mathbf{b}_s} (g'_i \ \vee \bigvee_{a_j \in A_n} g'_j)\Big)(\mathbf{x}) \quad = \quad \Big(\lozenge^{\mathbf{b}_s} (g_i \wedge \neg(\bigvee_{a_j \in A_n} \lozenge_k^{\mathbf{b}_s} g_j) \vee \bigvee_{a_j \in A_n} \lozenge^{\mathbf{b}_s} g'_j)\Big)(\mathbf{x})$$

  (induction hypothesis, distributivity of $\lozenge$ over $\vee$)

  $$= \quad \Big(\lozenge^{\mathbf{b}_s} (g_i \wedge \neg(\bigvee_{a_j \in A_n} \lozenge_k^{\mathbf{b}_s} g_j) \vee \bigvee_{a_j \in A_n} \lozenge^{\mathbf{b}_s} g_j)\Big)(\mathbf{x})$$

  (weakening of $\lozenge$, distributivity of $\lozenge$ over $\vee$, additivity of $\lozenge$)

  $$= \quad \Big(\lozenge^{\mathbf{b}_s} (g_i \vee \bigvee_{a_j \in A_n} g_j)\Big)(\mathbf{x}) \; .$$

By disjunction for all $a_i \in A_{n+1}$ and distributivity of $\Diamond$,

$$(\Diamond^{\mathbf{b}_s} \bigvee_{a_i \in A_{n+1}} g'_i)(\mathbf{x}) \quad = \quad (\Diamond^{\mathbf{b}_s} \bigvee_{a_i \in A_{n+1}} g_i)(\mathbf{x}) \ .$$

Since $\prec$ is a priority order and $|A|$ is finite, we have $T(s) = A_{|A|-1}$. ■

Proof of theorem 4.3

We show that any run of *TS* contains at least one action of each component. Since after the last of these actions the target state is an initial state of all components, we can iterate and thus prove that there are actually infinitely many occurrences of actions of each component in the run.

To show that any run involves at least one action of each component, consider the initial state $(s, \mathbf{x})$ with $s = (s_1, \ldots, s_n)$ of the run. Since *TS* is structurally live, we know that time can progress for any given amount, and that an action will be taken from this state. Let us show that whatever happens in *TS*, an action involving some given component, say TS_1, will be taken.

To show this we will consider the successors of $(s, \mathbf{x})$ in *TS*. Since the clock spaces of the components are distinct, we can decompose the valuations of *TS* in n valuations, each one associated to a component. In this proof we will denote by $\mathbf{x} = \mathbf{x}_1 : \ldots : \mathbf{x}_n$ such a decomposition. The successors of $(s, \mathbf{x})$ in *TS*, without taking any transition involving TS_1, are then of the form $((s_1, s'_2, \ldots, s'_n),\ \mathbf{x}_1 + d\mathbf{b}_1 : \mathbf{x}'_2 : \ldots : \mathbf{x}'_n)$, where $d \in \mathbb{D}$ is the elapsed time from $(s, \mathbf{x})$. Let us show that there exists a limit, $d_0 \in \mathbb{D}$, for such times. Due to structural liveness of the product, we know that time will progress beyond d_0, and thus that an action involving TS_1 will be taken.

Let $t = (s_1, a, s'_1)$ a transition of (TS_1, pr_1) with $h_1(t) = (g, \tau, r)$, $(\Diamond^{\mathbf{b}_{s_1}} g)(\mathbf{x}_1)$, and either $\tau = \delta$ and g is bounded, or $\tau = \epsilon$. Such a transition exists, since $(s_1, \mathbf{x}_1)$ is an initial state of (TS_1, pr_1), and (TS_1, pr_1) is locally livelock-free by theorem 4.1. If a is an interleaving action, then t is a transition of *TS* that will eventually become urgent and block time progress until *TS* executes an action of TS_1.

If a is a synchronizing action, the reasoning is somewhat more complex, since the component timed systems synchronizing with a can choose to execute transitions not involving a. For any control state $s = (s_1, \ldots, s_n) \in S$ there may be a synchronization transition $t^s = (s, \mathbf{a}, s')$ issued from s with $a \in \mathbf{a}$ and guarded by g_s. Moreover, there is an interleaving transition $t^s_1 = (s, a, s') \in T$ guarded by g_1. By definition 2.13, $\big(\Diamond^{\mathbf{b}_s}(g_1 \vee g_s)\big)(\mathbf{x}) \iff \big(\Diamond^{\mathbf{b}_1} g\big)(\mathbf{x}_1)$, that is, some action involving a will eventually become enabled in (TS, pr) whenever a will eventually become enabled in (TS_1, pr_1). Both t^s_1 and t^s are bounded, and delayable or eager. Therefore, for any control state of *TS* for which TS_1 is in state s_1 and for any clock valuation $\mathbf{x}$ such that $(\Diamond^{\mathbf{b}_1} g)(\mathbf{x}_1)$, either t^s_1 or t^s will eventually become urgent in (TS, pr) before $\neg\Diamond^{\mathbf{b}_1} g$. Moreover, g is bounded. Since (TS, pr) is structurally non-zeno, some action involving a will eventually be taken. ■

Proof of proposition 5.1

Let $s = (\ldots, s_j, \ldots, s_i, \ldots)$ be a product state with $j < i$ where (5.1) holds. Thus, $\forall r_1, r_2 \in R_n \;.\; U_{i,r_1} \;\wedge\; c_{r_1} \leqslant j < i \implies \neg U_{j,r_2}$.

If none of the processes P_i and P_j holds a resource in state s, then at most one of them will acquire a resource in any transition issued from s, since according to section 3.2.3, *BGN* actions are not allowed to synchronize.

If $U_{i,r_1}(s)$ with $c_{r_1} \leqslant j$, then P_j does not hold any resource by hypothesis. It follows that $\mathit{current}(j)(s) = j$, and $\widetilde{\mathit{blocking}}_{ij}(s)$. Therefore, P_j is prevented from acquiring a resource.

If, on the other hand, $U_{j,r_2}(s)$ (thus, $c_{r_2} \leqslant j < i$), then $\forall r_1 \in R_n \;.\; U_{i,r_1} \implies c_{r_1} > j$ by hypothesis. Therefore, $j < \mathit{current}(i)(s)$, which implies $\widetilde{\mathit{blocking}}_{ji}(s)$. Therefore, any resource allocation to P_i is prevented.

It follows that (5.1) is an inductive invariant. ∎

Proof of proposition 5.2

Since $\mathit{current}(j) \leqslant j$, it follows that $\widetilde{\mathit{blocking}}_{ij} \implies \mathit{blocking}_{ij}$. We show that $\mathit{blocking}_{ij} \implies \widetilde{\mathit{blocking}}_{ij}$. Suppose that $\mathit{blocking}_{ij}$, that is, $i \neq j$, and there is some $r_1 \in R_n$ such that $U_{i,r_1} \;\wedge\; c_{r_1} \leqslant j$. If $\forall k < j \;.\; \neg\widetilde{\mathit{blocking}}_{jk}$, then $\mathit{current}(j) = j$, and the implication follows immediately.

Otherwise, let k be minimal such that $k < j$ with $\widetilde{\mathit{blocking}}_{jk}$, say, $U_{j,r_2} \;\wedge\; c_{r_2} \leqslant \mathit{current}(k)$. Therefore, $c_{r_2} \leqslant \mathit{current}(k) \leqslant k < j$. It follows from equation (5.1) that P_k cannot hold a resource, and thus $\mathit{current}(k) = k$. Suppose that $k < c_{r_1}$. Then, $c_{r_2} \leqslant k < c_{r_1} \leqslant \min\{i, j\}$. Equation (5.1) implies that both $c_{r_2} \leqslant i < j$ and $c_{r_1} \leqslant j < i$ are impossible. This refutes the assumption $k < c_{r_1}$. It follows that $c_{r_1} \leqslant k$ and, because P_k is the process of minimal index with $\widetilde{\mathit{blocking}}_{jk}$, $\mathit{current}(j) = k$, which implies $\widetilde{\mathit{blocking}}_{ij}$. ∎

Proof of proposition 5.3

We show that $\widetilde{\mathcal{C}_{ij}^{CPU}} \iff \mathcal{C}_{ij}^{CPU}$.

$'' \implies''$: Suppose that $\widetilde{\mathcal{C}_{ij}^{CPU}} \;\wedge\; \neg\mathcal{C}_{ij}^{CPU}$. Notice that the term $\forall k < j \;.\; \neg\mathit{blocking}_{ik}$ implies that $\mathit{current}(i) \geqslant j \;\vee\; i < j$. Then we have

$$
\begin{aligned}
& \mathit{current}(i) < \mathit{current}(j) \;\wedge\; (j \leqslant i \;\vee\; \exists k \leqslant i \;.\; \mathit{blocking}_{jk}) \;\wedge \\
& (i \leqslant j \;\vee\; \forall l < j \;.\; \neg\mathit{blocking}_{il}) \\
\implies\; & \mathit{current}(i) < \mathit{current}(j) \;\wedge\; \big(j < i \;\wedge\; \forall k < j \;.\; \neg\mathit{blocking}_{il} \;\vee \\
& i < j \;\wedge\; \exists k \leqslant i \;.\; \mathit{blocking}_{jk} \;\vee \\
& \exists k \leqslant i \;.\; \mathit{blocking}_{jk} \;\wedge\; \forall l < j \;.\; \neg\mathit{blocking}_{il}\big) \\
\implies\; & \mathit{current}(i) < \mathit{current}(j) \leqslant j < i \;\wedge\; (\mathit{current}(i) \geqslant j \;\vee\; i < j) \;\vee \\
& \mathit{current}(i) < \mathit{current}(j) \leqslant i < j
\end{aligned}
$$

The first term of the last disjunction is clearly unsatisfiable. The second term implies according to the definition of *current* that both P_i and P_j hold a resource, say r_1 and r_2 respectively, and that there is a process P_k such that $c_{r_2} \leqslant k = \mathit{current}(j) \leqslant i < j$, which is impossible according to the invariant (5.1).
$'' \Longleftarrow''$: Suppose that $\mathcal{C}_{ij}^{CPU} \ \wedge \ \neg\widetilde{\mathcal{C}_{ij}^{CPU}}$. Then,

$$
\begin{aligned}
& (i < j \ \wedge \ \forall k \leqslant i \ . \ \neg \mathit{blocking}_{jk} \ \vee \ j < i \ \wedge \ \exists k < j \ . \ \mathit{blocking}_{ik}) \ \wedge \\
& \mathit{current}(j) \leqslant \mathit{current}(i) \\
\Longrightarrow \quad & \mathit{current}(j) \leqslant \mathit{current}(i) \leqslant i < j \ \wedge \ (\mathit{current}(j) > i \ \vee \ j \leqslant i) \ \vee \\
& \mathit{current}(j) \leqslant \mathit{current}(i) < j < i
\end{aligned}
$$

The first term of the last disjunction is unsatisfiable. The second term is, according to the same reasoning as above, in contradiction with the invariant (5.1). ■

Appendix B

The Tool PROMETHEUS

B.1 Error Messages

Each of the following error messages is completed by an identifier and the name of the file in which the error occurred.

- **Parse error at line n, near '*idf*'.**
- **END idf_1 does not match SYSTEM idf_2.**
- **END idf_1 does not match PND idf_2.**
- **END idf_1 does not match SCHEDULER idf_2.**
- **Contradictory resource declaration: *idf*.**
- **Multiple quantification: *idf*.**
- **Unknown process: *idf*.**
- **Undeclared place: *idf*.** Use of a place that has not been declared in the `places` statement.
- **Undeclared action: *idf*.**
- **Number of parameters mismatch: *idf*.**
- **Undeclared symbol: *idf*.**
- **Undeclared resource: *idf*.**
- **Undeclared parameter: *idf*.**
- **Multiply declared resource: *idf*.**
- **Multiply declared clock: *idf*.**
- **Multiply declared variable: *idf*.**

- **Feature not implemented: *idf*.**
- **Self-loop: *idf*.**
- **Invalid comparison: *idf*.**
- **Internal error: *idf*.** This error should never occur.
- **Parameter redeclaration: *idf*.**
- **Disabled action: *idf*.**
- **Undeclared clock: *idf*.** Use of a clock that has not been declared in the `clocks` statement.
- **Non time-convex eager guard: *idf*.**
- **Cannot open file: *idf*.**
- **blocking with 2 identical parameters: *idf*.**
- **undeclared variable: *idf*.**
- **undeclared guard: *idf*.**
- **multiply declared guard: *idf*.**
- **undeclared transition: *idf*.** Use of a transition that has not been declared in the `transitions` statement.
- **synchronization of two uncontrollable action: idf_1 with idf_2.**
- **restriction of uncontrollable action: *idf*.**
- **unused quantification: *idf*.**

B.2 Warning Messages

- **Warning: resource *idf*, formerly declared preemptable, now declared non-preemptable.**
- **Warning: priority circuit {*action_set*} in (*control state*) for [*region*].** Transitive closure of a priority order leads to one or more circuits consisting of the action set.
- **Warning: assertion failed for state n.** The property defined in the `assert` statement does not hold in state n of the generated system.
- **Warning: component '*idf*' uses undeclared resource '*idf*'. I consider it non-preemptable.**

- **Warning: no arrival action found.** This warning message is issued by the worst-case completion time calculus if some process has no arrival action, such that the minimal inter-arrival time cannot be computed.
- **Warning: synchronizing action '*idf*' was declared interleaving.** In the current implementation, synchronizing actions do not interleave in the generated IF file. This warning is issued if the action is not explicitly declared non-interleaving in the specification.

B.3 Abstract Grammars of the Input Files

In the following sections, **ident** denotes an identifier. $ident_a$, $ident_c$, $ident_f$, $ident_g$, $ident_p$, $ident_{pl}$, $ident_{pm}$, $ident_r$, and $ident_v$ stand for identifiers representing an action, a clock, a file, a guard, a process, a place, a parameter, a resource, and a variable, respectively.

B.3.1 Syntax of the System Description File

system ::= **SYSTEM ident** decl_resources templates syncs scheduler assert **END ident**
decl_resources ::= (decl_pr | decl_nonpr)*
decl_pr ::= **PREEMPTABLE** item$^+$ **;**
decl_nonpr ::= **NONPREEMPTABLE** item$^+$ **;**
item ::= $ident_r$
templates ::= ($ident_f$ components | imm_pnd)$^+$
components ::= (component **,**)* component
component ::= $ident_p$ **(** eff_params **)**
eff_params ::= ϵ | (**int ,**)* **int**
syncs ::= ϵ | **SYNCHRO** sync$^+$
sync ::= idents cut mode idents cut
mode ::= **MIN** | **MAX** | **MASTER** | **AND** | **OR**
cut ::= ϵ | **!**
imm_pnd ::= **PND ident (int)** pnd_body **END ident**
scheduler ::= ϵ | **SCHEDULER** $ident_f$ **(** eff_params **)**
assert ::= ϵ | **ASSERT** boolean_discr
boolean ::= **TRUE** | **FALSE** | $ident_{pl}$
| expression rel_op expression | boolean bool_op boolean
| **NOT** boolean | **(** boolean **)**
idents ::= $ident_a$ ('|' $ident_a$)*
zone ::= **TRUE** | **FALSE** | int rel_op $ident_c$ | $ident_c$ rel_op int
| int rel_op $ident_c$ **-** $ident_c$ | $ident_c$ rel_op $ident_c$
$ident_c$ **-** $ident_c$ rel_op int
int ::= **nat** | int arith_op int | **-** int | **(** int **)**

B.3.2 Syntax of the Scheduler Description File

scheduler	::=	**SCHEDULER ident (** params **)**
		opt_pnd nonpreemptable blocking_def prios **END ident**
params	::=	ϵ $\mid$ ($ident_{pm}$ **,**)* $ident_{pm}$
opt_pnd	::=	ϵ $\mid$ imm_pnd
nonpreemptable	::=	ϵ $\mid$ **NONPREEMPTABLE** $ident_r^+$
blocking_def	::=	pquant pquant blocking **:=** state_pred
blocking	::=	**BLOCKING (** $ident_p$ **,** $ident_p$ **)**
state_pred	::=	(quantifier **ident :** domain)* pred
quantifier	::=	**EXISTS** $\mid$ **FORALL**
domain	::=	**RESOURCE** $\mid$ **PREEMPTABLE**
		$\mid$ **NONPREEMPTABLE** $\mid$ **PROCESS**
pred	::=	blocking $\mid$ **USING (** $ident_p$ **,** $ident_r$ **)**
		$\mid$ **WAITING (** $ident_p$ **,** $ident_r$ **)**
		$\mid$ **TRUE** $\mid$ **FALSE** $\mid$ $ident_{pl}$ $\mid$ **[** abstract_region **]**
		$\mid$ ext_expression rel_op ext_expression $\mid$ pred bool_op pred
		$\mid$ **NOT** pred $\mid$ **(** pred **)**
ext_expression	::=	**nat** $\mid$ **CEILING (** $ident_r$ **)** $\mid$ **PRIO (** $ident_p$ **)**
		$\mid$ ext_expression arith_op ext_expression $\mid$ **-** ext_expression
		$\mid$ **$** $ident_{pm}$
prios	::=	ϵ $\mid$ **PRIORITIES** rule$^+$
rule	::=	[pquant] [pquant] $ident_p$ prio_op $ident_p$
		[**IF** state_pred]
		$\mid$ $ident_a$ prio_op $ident_a$ [**IF** state_pred]
pquant	::=	**FORALL** $ident_p$ **: PROCESS**
prio_op	::=	**<** **int** $\mid$ **<<**
int	::=	**nat** $\mid$ **$** $ident_{pm}$ $\mid$ int arith_op int $\mid$ **-** int $\mid$ **(** int **)**

where *abstract_region* is a *region* where the nonterminal *int* is replaced by *ext_expression*, and only clock differences may be constrained. Hence, an *abstract_region* is time-invariant (in absence of clocks).

B.3.3 Syntax of a Component Description File

pnd	::=	**PND ident (** params **)** pnd_body **END ident**
params	::=	ϵ | $(ident_{pm}$ **,**$)^*$ $ident_{pm}$
pnd_body	::=	places resources clocks vars guards trans
places	::=	**PLACES** $ident_{pl}{}^+$
resources	::=	(**RESOURCE** $ident_r$ **:** $ident_{pl}^+)^*$
clocks	::=	ϵ | **CLOCKS** $ident_c{}^+$
vars	::=	ϵ | **VARIABLES** $ident_v{}^+$
guards	::=	ϵ | **GUARDS** guard$^+$
guard	::=	$ident_g$ **:=** region
trans	::=	**INIT:** $ident_{pl}{}^+$ **;** transition*
transition	::=	$ident_a$ **:** (ϵ | **CTRL** | **UNCTRL**) in out condition
		trans_guard resets assigns **;**
in	::=	ϵ | **IN:** $ident_{pl}{}^+$
out	::=	ϵ | **OUT:** $ident_{pl}{}^+$
condition	::=	ϵ | **IF** boolean
expression	::=	**nat** | $ident_v$ | expression arith_op expression
		| **-** expression
arith_op	::=	**+** | **-** | ***** | **/** | **%**
bool_op	::=	**AND** | **OR**
trans_guard	::=	**GUARD** ($ident_g$ | region) type
type	::=	**EAGER** | **DELAYABLE** | **LAZY**
resets	::=	ϵ | **RESETS** $ident_c{}^+$
assigns	::=	ϵ | **LET** $ident_v$ **=** expression
region	::=	zone | region bool_op region | **(** region **)**
int	::=	**nat** | **$** $ident_{pm}$ | int arith_op int | **-** int | **(** int **)**
rel_op	::=	$<$ | $\leqslant$ | $=$ | $\geqslant$ | $>$

Index

Résumé :

La modélisation des systèmes temps-réel est nécessaire pour l'analyse de leur temporisation. Le modèle doit être une représentation fidèle du comportement dynamique et permettre un raisonnement compositionnel pour le traitement de systèmes complexes.

Nous présentons une méthodologie de modélisation basée sur une décomposition en couches du système temps-réel en un ensemble de processus — modélisés sous forme de systèmes temporisés —, une couche de synchronisation qui assure des propriétés fonctionnelles et un ordonnanceur spécifiant des politiques d'ordonnancement telles que RMS, EDF ou le *priority ceiling protocol*. Pour la synchronisation, comme pour l'ordonnanceur, la notion de priorités joue un rôle primordial pour permettre leur composabilité. La méthodologie utilise une opération de composition parallèle "souple" conçue pour préserver des propriétés de progression. Nous définissons la propriété de vivacité structurelle, qui est facile à vérifier et préservée par la composition parallèle et les priorités. Ceci permet un raisonnement compositionnel sur la vivacité et la construction de systèmes temporisés vivants.

L'outil de modélisation compositionnelle PROMETHEUS implante ces résultats et supporte la méthodologie proposée. Les processus temps-réel, leur synchronisation et leur ordonnanceur sont spécifiés dans un langage de haut niveau. PROMETHEUS analyse la sûreté, les propriétés temporelles et la cohérence de la spécification. Des traitements complémentaires peuvent être effectués grâce à la connexion de PROMETHEUS à la plate-forme de validation IF. Notre méthodologie est illustrée par des applications réelles faisant intervenir une synchronisation et un ordonnanceur non triviaux.

Abstract:

Modelling real-time systems is a prerequisite for timing analysis. Models should faithfully represent the dynamic behaviour, and allow compositional reasoning to cope with complexity.

We present a modelling methodology based on a layered decomposition of real-time systems into a set of processes — modelled as timed systems —, a synchronization layer ensuring functional properties, and a scheduler specifying scheduling policies such as RMS, EDF, or the priority ceiling protocol. For both the synchronization layer and the scheduler, priorities play an instrumental role to achieve composability. The methodology uses a "flexible" parallel composition operation designed to preserve progress properties. We define the property of structural liveness, which is inexpensive to verify, and preserved by flexible parallel composition and priorities. This allows compositional reasoning about liveness, and construction of live timed systems.

The compositional modelling tool PROMETHEUS implements these results and supports the proposed methodology. Real-time processes, their synchronization, and a scheduler are specified in a high-level language. Safety, timing properties, and consistency of the specification are analyzed. The model can be further analyzed thanks to a connection of PROMETHEUS with the validation platform IF. Our methodology is illustrated by real-world applications involving non-trivial synchronization and scheduling.