

# A Time Constrained Real-Time Process Calculus

Viktor Leijon

Luleå University of Technology  
Department of Computer Science and Electrical Engineering  
EISLAB



---

# A Time Constrained Real-Time Process Calculus

Viktor Leijon

EISLAB

Dept. of Computer Science and Electrical Engineering  
Luleå University of Technology  
Luleå, Sweden

---

**Supervisor:**

Johan Nordlander  
Jingsen Chen





*The night is darkening round me,  
The wild winds coldly blow;  
But a tyrant spell has bound me  
And I cannot, cannot go.*  
- Emily Brontë



---

# ABSTRACT

---

There are two important questions to ask regarding the correct execution of a real-time program:

- (i) *Is there* a platform such that the program executes correctly?
- (ii) Does the program execute correctly on *a particular* platform?

The execution of a program is correct if all actions are taken within their execution window, i.e. after their release time but before their deadline. A program which executes correctly on a specific platform is said to be *feasible* on that platform and an *incorrect* program is one which is not feasible on any platform.

In this thesis we develop a timed process calculus, based on the  $\pi$ -calculus, which can help answer these questions. We express the time window in which computation is legal by use of two time restrictions, **before** and **after**, to express a deadline and a release time offset respectively.

We choose to look at correctness through the traces of the program. The trace of a program will always be a sequence of interleaved internal reductions and time steps, because programs have no free names. We define the meaning of a feasible program by use of these traces. In particular we define the speed of a particular system as the ratio between work steps and time steps.

Based on this calculus we show how the two questions above can be answered in terms of traces of the process calculus and prove the classical utilization limit for Earliest Deadline First scheduling holds.



---

# CONTENTS

---

|  |    |
|--|----|
| CHAPTER 1 – INTRODUCTION                                 | 3  |
| 1.1 Problem Overview . . . . .                           | 3  |
| 1.2 Our Approach . . . . .                               | 4  |
| 1.3 Roadmap for the Thesis . . . . .                     | 5  |
| 1.4 Contributions . . . . .                              | 5  |
| CHAPTER 2 – BACKGROUND                                   | 7  |
| 2.1 Real-Time Systems . . . . .                          | 7  |
| 2.2 Scheduling Theory . . . . .                          | 10 |
| 2.3 Process Calculi . . . . .                            | 12 |
| 2.4 Real-Time Programming Languages . . . . .            | 15 |
| 2.5 Other Formalisms . . . . .                           | 17 |
| CHAPTER 3 – THE $Ti\pi$ CALCULUS                         | 19 |
| 3.1 The Choices . . . . .                                | 20 |
| 3.2 The Untimed Calculus . . . . .                       | 20 |
| 3.3 The Timed Calculus . . . . .                         | 23 |
| 3.4 A Few Examples . . . . .                             | 25 |
| 3.5 Properties of the Timed Calculus . . . . .           | 26 |
| 3.6 Extending the Calculus With a Time Blocker . . . . . | 27 |
| 3.7 Encoding Input-guarded Choice . . . . .              | 28 |
| 3.8 Time Durations and Parallelism . . . . .             | 29 |
| 3.9 Other Timed Process Calculi . . . . .                | 30 |
| 3.10 Discussion . . . . .                                | 33 |
| CHAPTER 4 – TIMED EQUIVALENCIES                          | 35 |
| 4.1 Definitions for Bisimilarities . . . . .             | 35 |
| 4.2 Some Equalities . . . . .                            | 37 |
| 4.3 The Admissibility of Optimizations . . . . .         | 38 |
| CHAPTER 5 – SCHEDULING AND TIMED PROCESSES               | 39 |
| 5.1 Classical Scheduling Theory . . . . .                | 40 |
| 5.2 Other Process Calculi and Scheduling . . . . .       | 43 |

|   |    |
|---|----|
| CHAPTER 6 – CONCLUSIONS AND FUTURE WORK                   | 45 |
| 6.1 Conclusions . . . . .                                 | 45 |
| 6.2 Future work . . . . .                                 | 45 |
| APPENDIX A – GLOSSARY                                     | 49 |
| APPENDIX B – DETAILED PROOFS                              | 53 |
| B.1 Proofs About the Properties of the Calculus . . . . . | 53 |
| B.2 Bisimilarities Proofs . . . . .                       | 57 |
| B.3 Scheduling Proofs . . . . .                           | 60 |

---

# PREFACE

---

This work is the result of over two years of meandering through computer science. I would like to take this opportunity to thank a few people without who I doubt I would ever have found a steady path.

First of all my advisor Johan Nordlander who has been wonderful at instilling in me a sense of what computer science is all about. I would also like to thank my co-supervisor Jingsen Chen for first introducing me to computational models.

Peter Jonsson shared an office with me all this time, and has been a constant guinea pig for immature and abandoned ideas, which probably benefitted me as much as it inconvenienced him. I would like to thank Martin Kero and Andrey Kruglyak for being part of useful discussions in the Timber group. I am also grateful for the support and companionship of the other graduate students at EISLAB.

Outside of work I would like to thank my family and friends for keeping me as sane as possible through this process. I know my time for you has been more limited than I would have liked it to.

My work has been funded by the European Union FP6 project SOCRADES (<http://www.socrades.eu/>) and by PhD-Polis in cooperation with Oulu University. I am also grateful for the support of the national Swedish Real-Time Systems research initiative ARTES (<http://www.artes.uu.se>), supported by the Swedish Foundation for Strategic Research.



### 1.1 Problem Overview

Real-time systems are increasingly a part of our everyday lives, and thus increasingly important to us. In fact, it has become so central that the design of embedded and real-time systems is considered one of the great challenges for Computer Science by some researchers (Henzinger and Sifakis 2006; Stankovic et al. 2005).

We start out with a definition of what we mean by a real-time system:

**Definition 1.1** (A real-time system). *A real-time system is a system in which the correctness of a program is dependent not only on its output, but also on time at which the output is produced.*

There are two important questions that we want to ask regarding the correct execution of a real-time program:

- (i) *Is there a platform such that the program executes correctly?*
- (ii) *Does the program execute correctly on a particular platform?*

One of our central issues will be the tension between wanting to make general statements about our requirements of a system and the desire to reason about specific platforms within the same framework. General statements are statements without reference to a particular platform such as “the action *ring bell* must execute every 10 time units”, while the specific statements are of the type “will the system run correctly on platform X?”

To be able to answer these kinds of questions we want to stand on some kind of formal foundation, we are looking for a way to combine the two types of questions.

The challenges facing us are many. For our work to be useful it must be both powerful enough to express real systems, and simple enough to make an analysis

feasible. We must be able to express both time and concurrent computations, and it must be possible for us to prove at least some interesting properties about what is expressed.

## 1.2 Our Approach

The problem as outlined in Section 1.1 has many different aspects, but in this thesis we will attempt to find a formalism that could serve as the basis for a programming language, one which is close enough to a way in which we want to write computer programs that conclusions drawn based on our formalism can be easily understood in terms of programs.

Our work takes the viewpoint of the programmer, we imagine that our terms do not *describe* the workings of a system but that they *prescribe* how the system should work. We expect that a programmer will write the terms in our calculus. Perhaps not directly, as we do in our examples in Section 3.4, but in some kind of sugared form. This is analogous to the way a functional programmer in Haskell can be said to write terms in some variant of the lambda calculus.

This sets the current work slightly apart from most previous work which has had as a goal to model existing systems, in some sense to *describe* their behavior. The terms in such calculi are descriptive, in that they describe the workings of a system.

The focus has often been on verification of properties, for instance via the use of real-time model checking. The present work has more modest aspirations in the field of verification, and we satisfy ourselves with examining the schedulability of processes (Chapter 5).

This should not be taken as criticism against real-time modeling or verification, only as a necessary limitation of our scope. However, it should be noted here that in some sense the program itself is a model of something.

We want to be able to formally reason about the properties of hard real-time systems, in particular properties relating to functional and temporal correctness. Unfortunately, many things that we take for granted in a sequential system, or in an untimed concurrent system, become more complicated in a timed concurrent system. The fundamental change is that in a real-time system there are constraints on *when* actions may occur.

This observation leads us away from assigning a fixed duration to each step and thus away from considering the execution of the program on a particular computer, instead we want to continue viewing the correct execution as a set of possible behaviors and only limit them by the explicit constraints given by the programmer.

What we want to do is to separate the *correctness* of a program from the *feasi-*

*bility* of scheduling the program on a particular computer, which opens up not only for classical sequential compiler optimizations, but for all kinds of optimizations of concurrent timed programs.

## 1.3 Roadmap for the Thesis

We will start by covering some background in Chapter 2. We then move on to define our calculus (Chapter 3) and bisimilarities for it (Chapter 4). Finally we discuss scheduling in Chapter 5. We will also cover related work towards the end of each chapter.

We will assume that the reader has some familiarity with real-time systems and process calculi, as well as computer science in general.

For an introduction to real-time systems there are many good text books, for instance the books by Kopetz (1997) or Burns and Wellings (2001). The ARTIST roadmap (Bouyssounouse and Sifakis 2005) gives a good overview of the current thrusts of research in real-time systems.

When it comes to process calculi in general and  $\pi$ -calculus in particular either the textbook by Sangiorgi and Walker (2003) or the one by Milner (1999) is recommended, both of which give solid introductions to the subject. For an introduction to timed process calculi in particular there is a good tutorial by Hennessy (1992).

The reader with ample pre-requisites on the other hand may skip Chapter 2 and go directly to the the new calculus in Chapter 3. The more practically inclined may prefer to start by looking at the examples in Section 3.4. There is a glossary in Appendix A which defines some important terms. Complete proofs are available in Appendix B.

The material in this thesis is based on an article which has been submitted to FoSSaCS 2009 (Leijon and Nordlander 2009).

## 1.4 Contributions

The main contributions of this thesis towards the development of a foundation for real-time systems are:

- We introduce a complete timed process calculus called  $Ti\pi$  based on the  $\pi$ -calculus (Chapter 3). The calculus has both baseline and deadline constraints on processes, and actions are durationless.
- Further we give a definition of bisimulation for the calculus, in Chapter 4, together with proofs for some useful bisimilarities including a motivation for why sequential optimizations can be allowed in the calculus.

- A useful schedulability analysis for the real-time process calculus is developed, based on classical scheduling theory in Chapter 5. This is achieved by introducing a judgement that we call  $N$ -feasibility on the traces of processes.

---

# CHAPTER 2

---

## Background

In the following chapters we will develop a calculus for real-time systems. To provide some background we first look at a few important concepts.

We start by discussing real-time system concepts in general and then we move on to scheduling and process calculi, the intersection of which is the core of our work in the coming chapters. Finally we look at real-time programming languages and other formalisms, which are alternative ways to approach the same type of problems.

### 2.1 Real-Time Systems

In the introduction we introduced and defined real-time systems. The essence of a real-time system is that it combines temporal and functional requirements, and of course dependability requirements.

Real-time systems are often considered in the same breath as embedded systems, and while there is a strong relationship between them they are by no means the same thing. To give an example, the ARTIST roadmap (Bouyssounouse and Sifakis 2005) is titled “Embedded Systems Design” and combines both real-time and embedded computing. We will not discuss the additional constraints imposed by embedded systems, but deal with real-time systems in general. In general our focus is on systems with more than one task, or process, where the tasks may interact with each other.

We will explain a few key concepts, but refer to any introductory text book on real-time systems for details (Kopetz 1997; Burns and Wellings 2001).

Normally we think of the behavior of untimed concurrent systems as the set of all possible behaviors. For instance, let us assume that we have a global variable

$i := 5$  and a system in pseudo code:

$$\begin{aligned} P_1 &= \text{compute}_m; i := 0; \\ P_2 &= \text{compute}_n; \text{print } i; \end{aligned}$$

where  $\text{compute}_m$  represents  $m$  steps of “internal” computation. We expect one of two possible outputs, either 0 or 5, depending on how the processes are interleaved.

In a real-time system, things are a little bit more complicated: We have processes with temporal restrictions; they cannot start before their *release-time* and must finish before their *deadline*. This creates a time window, outside of which we would not expect any computation to be performed.

When considering these systems *working* and *waiting* becomes two very different things and it is tempting to assign a duration to each computational step. To demonstrate the consequences of this we can modify the process  $P_2$  in our first example to contain waiting instead of computation

$$\begin{aligned} P_1 &= \text{compute}_m; i := 0; \\ P'_2 &= \text{wait}_n; \text{print } i; \end{aligned}$$

where  $\text{wait}_n$  represents  $n$  time units of waiting and  $\text{compute}_m$  requires  $m$  time units of processing time. If  $m < n$  then the output will always be 0 but if  $m > n$  it may instead be 5 (depending on how the processes are interleaved).

Note that while  $\text{wait}_n$  takes the same amount of time on all platforms, the time required for  $\text{compute}_m$  varies between platforms.

In a sequential program it is sound to replace a term with a *faster* but functionally equivalent one, however in this example we see that changing the amount of computation required in a process can change its result. Indeed, even without changing the processes, we can get the same effect by executing them on a faster computer. In fact, we can imagine a computer that is so much faster or slower than the original one that  $\text{compute}_m$  takes  $m'$  units of time, for any  $m'$ . This is exactly what we examine more formally in the coming chapters.

### 2.1.1 Mutual Exclusion

In practice, several processes may need to access shared resources – for example memory areas or hardware resources – and in these cases it is desirable to have mutual exclusion between accesses. Mutual exclusions limits the execution of the program by only allowing one process at a time to access the resource.

We give an example in pseudo code:

**Example 2.1** (No mutual exclusion). *We assume a shared variable  $i = 0$  and local variables  $j$  and  $k$ .*

$$P_1 = j := i; i := j + 1;$$

$$P_2 = k := i; i := k * 2;$$

*In this situation, the resulting value of  $i$  can be either 0, 1 or 2 depending on how the execution of the two processes are interleaved. For instance,  $i = 0$  occurs when the execution of  $P_2$  is interleaved so that it reads  $i$  before  $P_1$  writes it, but writes  $i$  after  $P_1$  does.*

To solve this kind of problem a wide variety of primitives for mutual exclusion and message passing have been introduced. A primitive, but useful, approach is to simply introduce a special type of variable, that we call  $S$ , and primitives  $\text{lock}(S)$  and  $\text{unlock}(S)$  such that  $\text{lock}$  will block execution of the caller until the the lock  $S$  is available. This is in essence what is known as a binary semaphore.

We can now implement the same example again, using mutual exclusion.

**Example 2.2** (Mutual exclusion). *In addition to the variables  $i, j, k$  from Example 2.1 we also assume a lock  $S$ .*

$$P_1 = \text{lock}(S); j := i; j := j + 1; i := j; \text{unlock}(S)$$

$$P_2 = \text{lock}(S); k := i; k := k * 2; i := k; \text{unlock}(S)$$

In this example only one of the processes may run the critical section at any time, if  $P_1$  manages to lock  $S$  first then  $P_2$  will be stuck at the  $\text{lock}$  operation until  $P_2$  performs  $\text{unlock}$ . Vice versa if  $P_2$  gets the lock first.

In the presence of locks deadlocks might occur, and we also have the possibility of running into the well-known priority inversion problem (Section 2.2.3). A deadlock situation is a situation where two or more tasks are both unable to proceed because they each hold a resource that the other requires. We demonstrate this with a simple example.

**Example 2.3** (Deadlock). *Assume that we have two separate locks  $S$  and  $T$ .*

$$P_1 = \text{lock}(S); \text{lock}(T); \text{compute}_n; \text{unlock}(T); \text{unlock}(S)$$

$$P_2 = \text{lock}(T); \text{lock}(S); \text{compute}_m; \text{unlock}(S); \text{unlock}(T)$$

*In this example, we can have the interleaving where first  $P_1$  takes its first step, locking  $S$ , and then  $P_2$  takes its first step, locking  $T$ . In this situation, both processes are stuck trying to lock a lock held by the other process. This means that  $P_1$  and  $P_2$  are deadlocked.*

Deadlock freedom is one of the most essential properties of a real-time system, and also one of the most popular ones to prove for a system.

## 2.2 Scheduling Theory

Scheduling theory deals with how to select the process that will get access to the processor or another limited resource. The task of the scheduling algorithm is to divide access to the resources such that all processes get the resources they need to complete successfully.

In the standard model, each task  $\tau_i$  is characterized by a deadline  $d_i$ , a period  $T_i$ , a run-time  $C_i$  and a priority  $p_i$ . More advanced theories such as offset scheduling (Palencia and Harbour 2003) take into account both the dependencies between tasks and the release time jitter. While the standard model assumes that all tasks are periodic there are theories which allow for non-periodic tasks.

Scheduling theory is divided into two groups: (i) deadline based scheduling where the priorities are computed at run time based on deadlines and (ii) priority based scheduling where the priorities are assigned statically.

We will discuss two representative scheduling algorithms, Earliest Deadline First and Rate Monotonic scheduling. For a more complete comparison of these two methods see the work by Buttazzo (2005). Finally we will take a brief look at the Stack Resource Protocol, which manages scheduling with mutual exclusion.

The canonical reference for modern scheduling theory for periodic processes is the article by Liu and Layland (1973), which introduced most of the basic concepts. An authoritative and comprehensive review of the last 25 years of scheduling theory is given by Sha et al. (2004).

### 2.2.1 Earliest Deadline First Scheduling (EDF)

As indicated by its name the Earliest Deadline First scheduler gives the process with the earliest absolute deadline the highest priority (Liu and Layland 1973). Note that this gives processes dynamic priorities, so that the priority increases as the deadline approaches.

When a process arrives to the scheduler it is placed in a queue, ordered reversely by absolute deadline. Each time a process finishes executing the waiting process with the lowest absolute deadline is started.

This exceedingly simple scheduling algorithm turns out to be surprisingly good; if we have no relative phase and deadlines coincide with periods ( $T_i = d_i$ ), then the task set of  $n$  tasks is schedulable by EDF if and only if the utilization bound:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1. \quad (2.1)$$

That is, under the given assumptions, we can schedule all task sets up to a permanent 100% processor load.

In practice, with overheads, varying deadlines and in particular with the introduction of mutual exclusion constraints, the utilization bound becomes less than 1.

### 2.2.2 Rate Monotonic Scheduling (RM)

In a fixed priority scheduler each recurring task is given a fixed priority “off-line” (Liu and Layland 1973). For the Rate Monotonic scheduler each task should be assigned a priority  $p_i$  so that if  $T_i < T_j$  then  $p_i > p_j$ . One way to do this is to simply assign the inverse period so that  $p_i = 1/T_i$ .

The utilization bound for RM scheduling is:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1), \quad (2.2)$$

where the limit of the utilization bound is approximately 0.69. This bound should be contrasted to Equation 2.1. It should be noted that some task sets with a utilization above this bound can still be scheduled by RM, so it is only a bound on the guaranteed schedulability. Further, RM is optimal among fixed priority schedulers.

Fixed priority schedulers are very popular in practice, and the majority of the commercial real-time operating systems implement some variant of this scheme.

### 2.2.3 Priority Inversion and the Stack Resource Policy

When a low priority task holds a resource needed by a high priority task, the high priority task is forced to suspend, waiting on the low priority task. In this situation a naïve scheduler may let a medium priority task preempt the low priority task. This priority inversion has the effect of allowing a medium priority task delay a high priority task even though they have no resources in common, something that is in general undesirable.

Under the additional constraints that mutual exclusion imposes (Section 2.1.1) scheduling becomes more complicated. Using regular schedulers means that a deadlock may occur between processes. One scheduling algorithm which can handle scheduling with mutual exclusion, taking the priority inversion problem into account, is the Stack Resource Policy (SRP) (Baker 1991).

The SRP algorithm needs to know in advance which tasks will use which resources. This knowledge is used to make sure that before a job is allowed to start it must first have available all the resources needed for any *higher priority* task. A resource is available if it is either locked by the job just starting or is unlocked. This condition ensures that no task can be forced to wait for more than one lower priority task.

## 2.3 Process Calculi

Lambda calculus (Barendregt 2000) has long been used as a formalism for reasoning about sequential programs, and has formed a basis for much of the research in programming languages. This is particularly true for functional programming languages, but for concurrent or parallel systems the choice of a calculus has been less distinct. Nevertheless, process calculi have emerged as one popular formalism.

Surveying all the work done on process calculi is a massive task, far outside the scope of this thesis. We will focus our attention to CCS, in the following section, and its successor the  $\pi$ -calculus (Section 3.2) because they are the foundation for our work in this thesis. The terms process calculus and process algebra are used interchangeably. The fact that we do not extensively discuss other calculi such as CSP, ACP or  $\mu$ CRL is simply a matter of priorities, we will however refer them for comparison when appropriate. There is a brief, but good, overview of the history of process calculi by Baeten (2005).

An untimed process calculus such as CCS can be used for studying concurrent systems, and for exploring the behavior of distributed systems. While we focus on time, the importance of untimed systems should in no way be dismissed, and in fact, the untimed formalism occurs as a kernel of most timed ones, including the one we develop in Chapter 3.

### 2.3.1 Calculus of Communicating Systems (CCS)

The Calculus of Communicating Systems was developed by Robin Milner. It was developed and used extensively throughout the 80s. See for instance the book by Milner (1989) for a comprehensive treatment of CCS.

As the name suggests, the idea is that the calculus should be able to express systems (processes) communicating with each other over named channels. In plain CCS the only form of communication is synchronization, no data can be sent over the channels.

In the following presentation we follow the style of Chen (1992, Sec 1.1.1). Assume that there is a set of names  $\mathcal{N}$  and co-names  $\overline{\mathcal{N}}$  so that for every  $\alpha \in \mathcal{N}$  there is an  $\overline{\alpha} \in \overline{\mathcal{N}}$ , and a special name  $\tau$ . These two sets together with  $\tau$  are the labels

$$\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}.$$

In addition to this we adopt the conventions that  $\overline{\overline{\alpha}} = \alpha$  and  $\overline{\tau} = \tau$ .

We let the names  $P$  and  $Q$  range over the set of processes, and the names  $\alpha, \beta$  range over the set of labels  $\mathcal{L}$ . The name  $X$  is a meta level process term. CCS processes can now be defined syntactically:

$$P ::= \text{NIL} \mid \alpha.P \mid P + Q \mid P \mid Q \mid P \setminus a \mid P[S] \mid X \mid \mu X.P$$

$$\begin{array}{c}
\frac{}{\alpha.P \xrightarrow{\alpha} P} \text{R-ACT} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q' \quad \alpha \neq \tau}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \text{R-INTER} \\
\\
\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \text{R-LEFT} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \text{R-RIGHT} \\
\\
\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{R-PARLEFT} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \text{R-PARRIGHT} \\
\\
\frac{P \xrightarrow{\alpha} P' \quad \beta \neq \alpha \wedge \bar{\beta} \neq \alpha}{P \setminus \beta \xrightarrow{\alpha} P' \setminus \beta} \text{R-REST} \qquad \frac{P \xrightarrow{\alpha} P'}{P[S] \xrightarrow{S(\alpha)} P'[S]} \text{R-SUBST} \\
\\
\frac{E\{\mu.X.E/X\} \xrightarrow{\alpha} P}{\mu X.E \xrightarrow{\alpha} P} \text{R-REC}
\end{array}$$

Figure 2.1: Operational semantics for CCS

The process NIL is the insensitive process that can perform no actions, only capable of idling. This corresponds to a dead process.

When a process is on the form  $\alpha.P$  it is said to consist of the prefix  $\alpha$  and the suffix  $P$ . Such a process can perform the action  $\alpha$  and then evolve into  $P$ . Note that the prefix  $\alpha$  is a label from  $\mathcal{L}$ , and that  $\alpha$  and  $\bar{\alpha}$  are considered to be connected to the same channel.

The form  $P + Q$  is called summation or external choice. A process on this form can act either as  $P$  or  $Q$ , but not both. Parallel composition is denoted  $P \mid Q$  and indicates that the processes  $P$  and  $Q$  run concurrently.

The restricted process  $P \setminus \alpha$  is the process  $P$  restricted so that the name  $\alpha$  is not visible outside of  $P$ , that is  $P$  is not allowed to take the action  $\alpha$ . For relabeling a process  $P$  using a relabeling function  $S$  we write  $P[S]$ , where  $S$  is a substitution on names such that  $\overline{S(\alpha)} = S(\bar{\alpha})$ .

We will often omit NIL when it is the suffix of a process. That is, we will write  $a.\bar{b}$  instead of  $a.\bar{b}.\text{NIL}$ .

The final two forms allow recursive definitions, stating that a process variable by itself is a process, and that the definition  $\mu X.P$  should be interpreted as the possibly recursive process  $X = P$ . A process is an agent if it does not contain any free instances of process variables.

We give the operational semantics for CCS in Figure 2.1 as a labeled transition system (LTS). An LTS is given as a set of states and a labeled transition relation between states. Transitions are given as  $P \xrightarrow{l} P'$ , interpreted as  $P$  goes to  $P'$

performing action  $l$ .

We consider  $\tau$  to be the empty, or internal, action while  $\alpha$  and  $\bar{\alpha}$  are the action and co-action synchronizing over the channel  $\alpha$ .

The rule R-INTER describes the interaction of two processes P and Q over a channel  $\alpha$ , which results in an internal action  $\tau$  of the composed system  $P \mid Q$ . Many authors, explicitly or implicitly, assume the use of *value-passing CCS* where the interaction also allows the exchange of values over the channel.

We finish with a classical example of a system described in CCS.

**Example 2.4** (The vending machine). *The canonical example by Milner (1989, p. 23) is the vending machine where one can pay either one penny and press the small button to get a small cup of coffee, or two pennies and press the big button to get a big cup of coffee. We present here a slightly extended variant of the original example that can be described like this in CCS, using recursive definitions:*

$$\begin{aligned} V_0 &= \overline{\text{penny}}.V_1 \\ V_1 &= \overline{\text{small}}.V_0 + \overline{\text{penny}}.V_2 \\ V_2 &= \overline{\text{small}}.V_1 + \overline{\text{big}}.V_0 \end{aligned}$$

*A customer who wants a big coffee can then be seen as the process:*

$$C_1 = \text{penny.penny.big},$$

*and the slightly more eccentric customer who wants three small coffees may be something like this:*

$$C_2 = \text{penny.penny.small.penny.small.small}.$$

*Finally the confused customer who wants a big coffee but only wants to pay a penny for it:*

$$C_3 = \text{penny.big}.$$

*We can study how the system composed of a vending machine and our first customer can develop, we annotate each transition with the rules used, the label will always be  $\tau$ :*

$$\begin{aligned} C_1 \mid V_0 &= \text{penny.penny.big} \mid \overline{\text{penny}}.V_1 \xrightarrow{\text{R-INTER}} \\ &\text{penny.big} \mid \overline{\text{small}}.V_0 + \overline{\text{penny}}.V_2 \xrightarrow{\text{R-INTER+R-RIGHT}} \\ &\text{big} \mid V_2 = \text{big} \mid \overline{\text{small}}.V_1 + \overline{\text{big}}.V_0 \xrightarrow{\text{R-INTER+R-LEFT}} \text{NIL} \mid V_0. \end{aligned}$$

*The confused customer on the other hand will have the following development:*

$$C_3 \mid V_0 = \text{penny.big} \mid \overline{\text{penny}}.V_1 \xrightarrow{\text{R-INTER}}$$

$$\text{big} \mid \overline{\text{small}}.V_0 + \overline{\text{penny}}.V_2$$

*And we can see that the system cannot take any additional action, and the customer will not get his big coffee.*

## 2.4 Real-Time Programming Languages

There has been quite a lot of interest in developing special purpose programming languages for real-time systems, or variants of existing programming languages to make them more suitable for real-time systems.

We will look at some of real-time languages and their properties. While real-time programming languages have advantages when it comes to analyzing their real-time properties, general-purpose analysis techniques, such as the explicit time approach suggested by Lamport (2005), can be applied to just about any programming language.

### 2.4.1 Special Languages

We will first consider the synchronous languages that have been constructed especially for real-time programming. These languages are based upon timed automatas, relying on this well studied formalism for theoretical foundations.

The synchronous languages work by defining equations which are computed synchronously, either on the arrival of an external event or on a clock tick. The most well-known synchronous languages are the French languages Esterel, Signal and Lustre. Benveniste et al. (2003) gives a good overview of the development of the synchronous languages, we will take a brief look at Lustre.

Lustre is based on the equation-centric style of control engineering and views the program as a set of flows (equations) which are evaluated at each clock tick. For each flow  $f$  they have an operator  $\text{pre}(f)$  which gives the value of the flow in the previous time instant and  $\rightarrow$  which constructs a flow with an initial value.

An example from Benveniste et al. (2003) is the following small program:

```
edge = false -> (c and not pre(c))
edgecount = 0 -> if edge then pre(edgecount) + 1
                else pre(edgecount)
```

The flow `edge` starts out as false at the first instant, in the following instants it becomes true if some other flow `c` has changed from false to true. The second flow `edgecount` increases if the flow `edge` is true, counting the number of edges.

The synchronous languages allow for programs which look more like the equations that control engineers work with, at the cost of being a bit less natural for a programmer.

Giotto (Henzinger et al. 2003) on the other hand is geared towards describing models rather than writing the whole real-time system, attempting to bridge the gap between the software engineer and the control engineer.

Giotto is based on describing the system in as the communication between independent, periodically triggered, tasks and is inspired by the Time Triggered Architecture (Kopetz and Bauer 2003), which is based on a similar premise.

The Timber programming language (Black et al. 2002; Carlsson et al. 2003) is a functional programming language based on reactive objects with both synchronous and asynchronous communication. Timber has built-in deadlines/baselines and much of the motivation behind the development of this calculus has been to create a timed calculus that can be used in the Timber project as a help for the further development.

## 2.4.2 Real-Time Extensions to “Normal” Languages

There are several “regular” languages which are used for the construction of real-time programs, perhaps three the most common ones are C, Ada and Java.

Ada 95 supports real-time programming through “Annex D”, an extension giving real-time support. It provides protected objects, fixed priority scheduling and absolute/relative delays and other features extending the core language to make it more suitable for real-time programming (Bouyssounouse and Sifakis 2005, Ch. 25).

The Real-Time Specification for Java (RTSJ) was developed by the Real-Time for Java Experts Group (Bollella et al. 2000) and was intended to bring real-time properties to the Java programming language, in essence by extensions in the same spirit as those contained in Annex D for Ada 95. The summary by Bollella and Gosling (2000) explains the design philosophy.

The RTSJ gives backwards compatibility, introducing no changes to the core Java language, but instead focusing on adding predictability in areas such as scheduling, memory management and resource sharing. This allows for a very familiar programming environment for existing programmers, while giving less stringent real-time properties than those given for instance by the synchronous languages.

## 2.5 Other Formalisms

Process calculus is far from the only possible formalism that can be used to describe a real-time system, but perhaps the one that holds the most interest from a programming language point of view.

Other formalisms have often been used either for verification of abstract models, or for code generation of “templates” where the programmer is expected to fill in the fiddly bits himself.

We will take a brief look at some other formalisms that can be used to represent a timed system, and what they can be used for.

### 2.5.1 Timed Automatas

Timed automatas are an extension of Büchi state automata with timing constraints. For an introduction to general automata theory see e.g the textbook by Lewis and Papadimitriou (1998). Timed automatas were introduced by Alur and Dill (1994) and have become very popular for verification and modeling.

Several popular model checkers use variants of timed automatas as the underlying formalism. One example is UPPAAL (Larsen et al. 1997), and there is a fairly recent tutorial style article by Bengtsson and Yi (2003) which explains the algorithms behind the model checking.

### 2.5.2 Real-Time Logic

One mathematically very attractive solution to the modeling problem is to model the real-time system using a strict logical framework. This allows us to draw on a great body of existing knowledge about logic and the verification of logic formulae.

The attraction for many of these logics is that they combine relative power and mathematical clarity with efficient model-checking (Henzinger et al. 1992).

It is worth noting that in some instances a timed process calculi can be related directly to a timed logic. For instance Yi (1991, ch. 8) relates his timed CCS with the modal Hennessy-Milner logic (Hennessy and Milner 1985). This kind of approach opens up the possibility to combine logic and process calculi when looking for equivalences between terms.

There are many different real-time, or temporal, logics. For an overview, see the article by Henzinger (1998). There is also the duration calculus, a calculus based on interval logic (Hansen and Chaochen 1997), using the notation of state and the duration of states as the basis for modeling.

### 2.5.3 Model Checking

Model checking is not a single algorithm, but rather a name for a certain type of techniques. The basic idea of model checking is to examine all possible “executions” of a model, and verifying that none of them violate any of a set of desired correctness properties. Models are often specified as timed automatas or directly in some form of real-time logic.

There are some platforms that implement model checking for real-time systems, of these we will have already mentioned the UPPAAL (Larsen et al. 1997) model checker, which seems to be one of the more expressive model checkers for real-time systems (Lamport 2005, sec. 6.4).

Model checking has the advantage that practically any property can be checked, but since the size of the state space is in general exponential of the size of the system (Henzinger et al. 1992), this “state explosion” means that verification can easily make it prohibitively costly to perform model checking. Most problem for timed model checking are known to be PSPACE-hard (Henzinger et al. 1992).

While the prospect of verifying arbitrary properties of real-time systems is a very exciting one, Lamport (2005, p. 6) has a bleaker view:

The earliest published real-time verification method that I know of, by Bernstein and Paul K. Harter (1981), was an implicit-time approach based on a toy programming language. They presumably hoped that their method could be extended to verify actual programs. However, the difficulties encountered in trying to extend ordinary program verification to actual programs makes such a hope now seem naive. Most recent work on real-time verification has been directed toward higher-level descriptions of algorithms, protocols, or system designs.

Despite this, model checking has many practical uses, see for instance the UPPAAL pamphlet (UPPAAL 2007) for examples of applications and case-studies.

---

## CHAPTER 3

---

# The $Ti\pi$ calculus

We started out in Section 1.2 by discussing what we consider to be a suitable calculus for real-time systems. In this chapter we will develop such a calculus formally, starting from the  $\pi$ -calculus. Since the result is a timed variant of the  $\pi$ -calculus we call our development the  $Ti\pi$ -calculus.

What we wanted was a formalism where we could express the timed behavior in a machine independent manner, so that we can make judgements which do not depend on any particular platform. At the same time, we would like it to be *possible* to discuss the correctness of a program on a certain platform as well.

This leads us away from assigning a fixed duration to each step and thus away from considering the execution of the program on a particular computer. Instead we want to continue viewing the correct execution as a set of possible behaviors and only limit them by the explicit constraints given by the programmer.

What we want to do is to separate the *correctness* of a program from the *feasibility* of scheduling the program on a particular computer, which opens up not only for classical sequential compiler optimizations, but for all kinds of optimizations of concurrent timed programs. This is a calculus with *instantaneous actions*, which means that all atomic actions take zero time, and thus any sequence of atomic actions take zero time. Time progress, which we denote by special  $\delta$ -actions, is only related to certain timing constructions, the restrictions.

We choose to look at correctness through the traces of the program. The trace of a program will always be a sequence of alternating internal  $\tau$ -actions and time progressing  $\delta$ -actions, because a program has no free names. We define the meaning of feasible programs by use of these traces, in particular we define the speed of the system as the ratio between work steps and time steps.

We start by defining the  $Ti\pi$ -calculus in this chapter and will look at feasibility and scheduling in Chapter 5.

### 3.1 The Choices

As we will see in Section 3.9 many timed process calculi already exist, each with a slightly different approach to the core questions. In some sense the answers to the questions are the choices which define a timed calculus and it is our focus on the combination of the generic and the specific that has directed our choices.

These are the five core questions, within parenthesis are the choices made in our calculus. They are far from the only important questions, but they are ones often mentioned, and they capture a lot of the essence of a timed calculus. Similar summaries of choices can be found for instance in the articles by Nicollin and Sifakis (1991, sec. 2.3) and Hennessy (1992, Lemma 2.3-2.6).

- (Q1) **Is time flow deterministic?** A system is time deterministic if only atomic actions (that is, actions that we associate with work being performed) can introduce non-determinism, while time progress itself cannot. (*Yes.*)
- (Q2) **Do we obey the maximum progress principle?** We say that we obey the maximum progress principle if a process will never idle when it can instead take an atomic action. This is analogous to “work-conserving” schedulers. (*No.*)
- (Q3) **What are our primitive operations?** Most calculi or systems use only one, or possibly a few, primitives for handling time. Examples include delay operators, timers, deadlines or time shift operators. (*A baseline/deadline restriction.*)
- (Q4) **Do atomic actions have duration?** Some calculi assign constant durations to atomic actions, while others allow instantaneous actions. (*No.*)
- (Q5) **Are processes patient?** If there is no other action to take, are all processes allowed to idle? (*Only within their deadlines.*)

### 3.2 The Untimed Calculus

The  $\pi$ -calculus is a further development of CCS which introduced the mobility of names. It was developed around ten years after CCS by Robin Milner, Joachim Parrow and David Walker (Milner et al. 1992a,b).

The basis for our work is the regular synchronous  $\pi$ -calculus, and our presentation follows that of Sangiorgi and Walker (2003), including the treatment of recursion.

Let  $N$  be a set of names, and construct the set of labels  $\Lambda = \{a, \bar{a} \mid a \in N\}$ . Our actions are  $\Lambda \cup \{\tau, \delta\}$  where we have two distinct actions  $\tau$ , the *internal action*, and

$\delta$ , the *time-progress action* which we discuss in the next section. We let  $\alpha$  range over the untimed actions  $\Lambda \cup \{\tau\}$  and  $\beta$  range over the timed actions  $\Lambda \cup \{\tau, \delta\}$ . We use  $\tilde{x}$  to denote a list of names.

We have removed replication in favor of recursion. We have also removed the mixed choice sum. It is possible to recover input-guarded choice and conditionals by encoding them in the calculus, see Section 3.7.

Free names, bound names and substitutions are defined in the usual way. We use the notation  $x.P$  to mean  $(\nu z) x(z).P$  when  $z \notin \text{fn}(P)$ . We will sometimes refer to a process without free names as a *program*. As a simplification we will leave out  $[\tilde{x}]$  in the cases where  $\tilde{x}$  is the empty list. Note that we leave out the inactive process when it is used as a suffix, so that we for example write  $x$  instead of  $x.\mathbf{0}$ .

We give the syntax for the basic calculus in the following definition and the operational semantics are in Figure 3.1.

**Definition 3.1** ( $\pi$ -calculus). *We use the following prefixes:*

$$\pi ::= \bar{x}y \mid x(z) \mid \tau$$

*Using the prefixes above we have the following definition for processes:*

$$P, Q ::= \mathbf{0} \mid P \mid Q \mid (\nu z) P \mid \pi.P \mid P \mid K[\tilde{a}].$$

*We also have process constant definitions:*

$$K \triangleq (\tilde{x}).P$$

*We refer to  $\mathbf{0}$  as the inactive process.*

**Example 3.2** (A  $\pi$ -vending machine). *The vending machine from Example 2.4 can of course be expressed in the  $\pi$ -calculus. Not having the sum operator forces us to a slightly awkward lock-based coding<sup>1</sup>. Note that we also use equality guards, which are not part of our calculus.*

---

<sup>1</sup>Technically, the coding is valid only for an asynchronous calculus, we omit a description of what would be required for a general encoding into a synchronous language. See the paper by Nestmann (2000, sect 3.1) for a more complete account.

$$\begin{array}{c}
\frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \text{OUT} \quad \frac{}{x(z).P \xrightarrow{xy} P\{y/z\}} \text{INP} \\
\\
\frac{}{\tau.P \xrightarrow{\tau} P} \text{TAU} \quad \frac{K \triangleq (\tilde{x}).P \quad P\{\tilde{a}/\tilde{x}\} \xrightarrow{\beta} P'}{K[\tilde{a}] \xrightarrow{\beta} P'} \text{CONST} \\
\\
\frac{P \xrightarrow{\alpha} P' \quad \text{bn}(\alpha) \cap \text{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{PAR-L} \\
\\
\frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \text{COMM-L} \\
\\
\frac{P \xrightarrow{\bar{x}(z)} P' \quad Q \xrightarrow{xy} Q' \quad z \notin \text{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\nu z) (P' \mid Q')} \text{CLOSE-L} \\
\\
\frac{P \xrightarrow{\beta} P' \quad z \notin n(\alpha)}{(\nu z) P \xrightarrow{\beta} (\nu z) P'} \text{RES} \quad \frac{P \xrightarrow{\bar{x}z} P' \quad z \neq x}{(\nu z) P \xrightarrow{\bar{x}(z)} P'} \text{OPEN}
\end{array}$$

Figure 3.1: Transition rules for the basic  $\pi$ -calculus

We divide it into several definitions for readability:

$$\begin{aligned}
V_0 &\triangleq \text{penny}.V_1 \\
V_1 &\triangleq (\nu l) (\nu t) (\nu f) \bar{l}t \mid \\
&\quad \left( \text{penny}.lv. ([v = t] (V_2 \mid \bar{l}f)) \mid [v = f] (\bar{l}f \mid \overline{\text{penny}}) \right) \mid \\
&\quad \left( \text{small}.lv. ([v = t] (V_0 \mid \bar{l}f)) \mid [v = f] (\bar{l}f \mid \overline{\text{small}}) \right) \\
V_2 &\triangleq (\nu l) (\nu t) (\nu f) \bar{l}t \mid \\
&\quad \left( \text{small}.lv. ([v = t] (V_1 \mid \bar{l}f)) \mid [v = f] (\bar{l}f \mid \overline{\text{small}}) \right) \mid \\
&\quad \left( \text{big}.lv. ([v = t] (V_0 \mid \bar{l}f)) \mid [v = f] (\bar{l}f \mid \overline{\text{big}}) \right)
\end{aligned}$$

### 3.3 The Timed Calculus

We extend the regular untimed reductions defined in Section 3.2 with a notion of time, the  $\delta$ -steps. Together with time we introduce restrictions on time, so that processes can only progress within their proper time-window.

We start with the new syntax:

**Definition 3.3** (*Ti* $\pi$ -calculus). *We use the following prefixes:*

$$\pi ::= \bar{x}y \mid x(z) \mid \tau$$

*The timed calculus extends processes with time restrictions. This gives the timed processes, using the prefixes above:*

$$P, Q ::= \mathbf{0} \mid P \mid Q \mid (\nu z)P \mid \pi.P \mid P \mid K[\tilde{a}] \mid \mathbf{before}_t P \mid \mathbf{after}_t P$$

where  $t$  is a time expression. We also have process constant definitions:

$$K \triangleq (\tilde{x}).P$$

We refer to  $\mathbf{0}$  as the inactive process.

We have two basic constructions for specifying the timed properties of processes:

**after<sub>t</sub> P:** Until  $t$  time-units have passed no work can be done in  $P$ , nor does any time pass within  $P$ . Furthermore,  $P$  is isolated from any deadlines not imposed by  $P$  itself.

**before<sub>t</sub> P:** Requires  $P$  to be finished, reduced to  $\mathbf{0}$ , within  $t$  time-units. Since we allow the relative deadline  $t$  to progress to 0 this should more properly be called 'not after  $t$ ' or 'before or at  $t$ ', but we judge these names a bit unwieldy and stick to **before<sub>t</sub>**.

#### 3.3.1 Time Expressions

We use the discrete time domain  $\mathcal{T} = \mathbb{Z} \cup \{\infty\}$  in this paper. We let  $\infty$  stand for an unlimited interval of time. Currently, no proofs depend on the fact that time is discrete, except for the determinism of time which would have to be rephrased slightly for continuous time. All our time expressions  $t$  are constants from  $\mathcal{T}$ .

|  |  |
|--|--|
| <b>Rules handling time constructs:</b>   |  |
| $\frac{}{\text{before}_t (\text{after}_t P) \xrightarrow{\tau} \text{after}_t P} \text{BREAKFREE}$             |  |
| $\frac{}{\text{before}_t \mathbf{0} \xrightarrow{\tau} \mathbf{0}} \text{DONE}$                                |  |
| $\frac{P \xrightarrow{\alpha} P'}{\text{before}_t P \xrightarrow{\alpha} \text{before}_t P'} \text{RESBEFORE}$ | $\frac{P \xrightarrow{\alpha} P'}{\text{after}_0 P \xrightarrow{\alpha} \text{after}_0 P'} \text{RESAFTER}$                |
| <b>Time progress rules:</b>  |  |
| $\frac{P \xrightarrow{\delta} P'}{\pi.P \xrightarrow{\delta} \pi.P'} \text{T-SEQ}$                             | $\frac{P \xrightarrow{\delta} P' \quad Q \xrightarrow{\delta} Q'}{P \mid Q \xrightarrow{\delta} P' \mid Q'} \text{T-PAR}$  |
| $\frac{}{\mathbf{0} \xrightarrow{\delta} \mathbf{0}} \text{T-NULL}$  | $\frac{b > 0}{\text{after}_t P \xrightarrow{\delta} \text{after}_{t-1} P} \text{T-WAIT}$                                   |
| $\frac{P \xrightarrow{\delta} P'}{\text{after}_0 P \xrightarrow{\delta} \text{after}_0 P'} \text{T-READY}$     | $\frac{P \xrightarrow{\delta} P' \quad t > 0}{\text{before}_t P \xrightarrow{\delta} \text{before}_{t-1} P'} \text{T-RUN}$ |

*Figure 3.2: Transition rules for  $Ti\pi$ -calculus*

### 3.3.2 Timed Semantics

The new semantic rules for time are presented in Figure 3.2, they form the full semantics for the language together with those in Figure 3.1. The first rules are related to the usual labeled transition rules for the  $\pi$ -calculus, but deal with time constructions:

**BREAKFREE:** States that an  $\text{after}_t$  clause is not affected by external deadlines, but can break free of such a context.

**DONE:** Indicates that a  $\text{before}_t$  restriction can be removed if the restricted process is reduced to the inactive process.

**RESBEFORE and RESAFTER:** These rules tell us that reduction may progress “as usual” either under the  $\text{before}_t$  or  $\text{after}_0$  restrictions, indicating that these are not restrictions on reductions at all.

Finally we have an additional six rules which deal with time progress:

**T-SEQ, T-PAR T-NULL, T-RES and T-CONST:** Simple rules which show that the untimed constructions do not affect time-flow.

**T-WAIT and T-READY:** Tells us that an  $\mathbf{after}_t$  blocks time progress in  $P$  until  $t$  reaches 0 at which time it becomes unblocked.

**T-RUN:** Time progress under a  $\mathbf{before}_t$  restriction is allowed as long as the deadline  $t$  does not go below 0.

### 3.4 A Few Examples

We take a few examples of interesting constructions, mainly from other authors, and examine them in our calculus.

**Delay.** Delay operators are common in timed process calculi, Berger (2002) codes a delay operator in terms of a timer with a timeout action and Moller and Tofte (1990) has an equivalent *loose time prefix*. We can define a delay operator similarly:

$$\Delta_{t,P} \triangleq \mathbf{after}_t P.$$

We can also expand this to a construction that performs an action at a specific time, similar to the atomic action at a time stamp that is primitive in  $\text{ACP}\rho$  (Baeten and Bergstra 1991):

$$@_{t,P} \triangleq \mathbf{after}_t \mathbf{before}_0 P.$$

**Cyclic.** Berger (2002) also describes a cyclic process as  $(\nu x) \bar{x} \mid !x.\mathbf{delay}^t(P|\bar{x})$ . We can similarly define a cyclicity operator

$$C_{t,P} \triangleq P \mid \mathbf{after}_t (C_t[P])$$

A simple ticker sending repeated ticks over the channel  $tick$  with period  $t$  could be built like this:

$$Ticker \triangleq C_{t,(\mathbf{before}_t \bar{tick})}$$

**Timers.** Timers with a timeout action are atomic in the calculus proposed by Berger (2002), as  $\mathbf{timer}^t(x.P, Q)$ , and in ATP (Nicollin and Sifakis 1994) as  $\lfloor P \rfloor^t(Q)$ . We cannot directly express this, but can implement something similar if we use an encoding of the input guarded choice:

$$\mathbf{timer}^t(x.P, Q) \triangleq (\nu q) \mathbf{before}_t (x.P + q.Q) \mid @_{t,\bar{q}}.$$

Note that this type of timers expose a problem, a timer in the sense of Berger (2002) sets a timeout only for the very first step in a process, while we prefer to talk about deadlines for a whole action. If what is desired is actually just about the reception of that first message, it can be rewritten slightly to move the body  $P$  outside the restriction:

$$\text{timer}^t(x.P, Q) \triangleq (\nu q) (\nu r) \text{before}_t(x.\bar{r} + q.Q) \mid @_{t,\bar{q}} \mid r.P$$

**Time-locks.** It is of course possible for processes to be deadlocked because of contradicting timing constraints. One pathological example is the process

$$(\nu z) (\text{before}_{10} \bar{x}y.P \mid \text{after}_{11} x(z).Q),$$

which would run well in an untimed system, but is stuck in our timed calculus.

### 3.5 Properties of the Timed Calculus

We will now take a look at some of the properties of the  $Ti\pi$ -calculus. The following properties seem to be almost universally considered important for timed systems (Nicollin and Sifakis 1991; Hennessy 1992), and give some intuition for how the system behaves.

We start with three properties which *do* hold. A common property is something called Time Persistence, Lemma 3.9, but we can actually prove an even stronger property, Time Confluence:

**Lemma 3.4** (Time Confluence). *The  $Ti\pi$ -calculus is time confluent: meaning that time progress can never remove an opportunity for a reduction step and reduction can never remove an opportunity for a time step.*

*This is equivalent to the following statement:*

$$\forall R, R', S : R \xrightarrow{\alpha} R' \wedge R \xrightarrow{\delta} S \Rightarrow S \xrightarrow{\alpha} R' \wedge R' \xrightarrow{\delta} S.$$

Which can be visualized in the following commutative diagram:

$$\begin{array}{ccc} R & \xrightarrow{\delta} & S \\ \alpha \downarrow & & \downarrow \alpha \\ R' & \xrightarrow{\delta} & S' \end{array}$$

*Proof.* See Appendix B. □

**Lemma 3.5** (Time Determinism). *The progress of time is deterministic:*

*If  $P \xrightarrow{\delta} P'$  and  $P \xrightarrow{\delta} P''$  then  $P'$  is syntactically identical to  $P''$*

*Proof.* The choice of reduction rule is uniquely determined by the structure of  $P$ . We proceed by induction on the structure of  $P$ :

*Case 0:* (the inactive process) using T-NULL.

*Case  $P \mid Q$ :* using RT-PAR on  $P$  and  $Q$ , and it holds by induction for  $P$  and  $Q$ .

*Case  $(\nu z) P$ :* using RES, and by induction it holds for  $P$ .

*Case  $\pi.P$ :* using T-SEQ, and by induction it holds for  $P$ .

*Case  $K[\tilde{a}]$ :* by CONST, and then it holds by induction for the result.

*Case  $\text{before}_t P$ :* using T-RUN, and then by induction for  $P$ .

*Case  $\text{after}_t P, t > 0$ :* using T-TIME.

*Case  $\text{after}_0 P$ :* using T-READY and then induction for  $P$ .

□

**Fact 3.6** (Time-progress preserves names). *If  $P \xrightarrow{\delta} P'$  then  $\text{fn}(P) = \text{fn}(P')$  and  $\text{bn}(P) = \text{bn}(P')$ .*

Finally we have two common properties, patience and maximum progress, which do not hold in the current calculus. The reason that we have neither patience nor maximum progress is that the  $\text{before}_t$  prefix provides a form of local urgency, rather than the global urgency. In particular the maximum progress property is unreasonable in a situation where we want to model real systems without assigning duration to actions, since it would force us to perform all actions immediately, without time progress.

**Definition 3.7** (Patience). *A patient system allows time to pass if there is no work to do. If  $P \not\rightarrow$  then  $P \xrightarrow{\delta}$ .*

**Definition 3.8** (Maximum Progress Property (MPP)). *Under the MPP, a process will never wait if it can perform work. If  $P \rightarrow P'$  then  $P \xrightarrow{\delta}$ .*

## 3.6 Extending the Calculus With a Time Blocker

In the general calculus presented in Section 3.3 there is no way to stop time progress, time will progress in all parts of the processes with equal speed. This is often the desired behavior, but sometimes we may prefer to start counting time after some particular event takes place

For this we extend our syntax of terms with a time blocking construct, which we call **now** :

$$\boxed{
\begin{array}{c}
\frac{}{\text{now } P \xrightarrow{\tau} P} \text{START} \\
\frac{}{\text{now } P \xrightarrow{\delta} \text{now } P} \text{T-BLOCKED}
\end{array}
}$$

Figure 3.3: The rules for extending  $Ti\pi$  with `now` .

$$P, Q ::= \mathbf{0} \mid P \mid Q \mid (\nu z) P \mid \pi.P \mid P \mid K[\tilde{a}] \mid \\
\text{before}_t P \mid \text{after}_t P \mid \text{now } P$$

We introduce two new rules to deal with `now` in Figure 3.3:

**START:** Lets us drop a `now` restriction to enable the underlying process is started.

**T-BLOCKED:** A process under a `now` restriction is blocked from time progress until the restriction is removed.

In essence `now`  $P$  creates a new baseline, insulating  $P$  from time progress until actual execution reaches it. The presence of `now` does not invalidate any of the properties of the calculus that we have already established, except for time confluence (Lemma 3.4) which is reduced to time persistence:

**Lemma 3.9** (Time Persistence). *The  $Ti\pi$ -calculus with `now` is time-persistent; meaning that time progress can never remove an opportunity for a reduction step.*

*This is equivalent to the following statement:*

$$\forall R, R', S : R \xrightarrow{\alpha} R' \wedge R \xrightarrow{\delta} S \Rightarrow \exists S'' : S \xrightarrow{\alpha} S''$$

*Proof.* Full proof in Appendix B. □

### 3.7 Encoding Input-guarded Choice

There has been a lot of research in the expressiveness of process calculi, and the role played by choice (Nestmann 2000; Nestmann and Pierce 1996). We can use an encoding inspired by the combination input-guarded choice as used by Nestmann and Pierce (1996) with a style for encoding the conditionals which is similar to that used by Pierce (1997).

The encoding of input guarded choice for an asynchronous fragment of the

language is (assuming  $q, v$  and  $t$  are neither in  $P$  nor in  $Q$ ):

$$\begin{aligned}
 x(u).P + y(v).Q \stackrel{def}{=} & (\nu q) (\nu t) (\nu f) \bar{q}t \\
 & \left| \left( x(u).q(b).(\bar{b} | t.(P|\bar{q}f) | f.(\bar{q}f | \bar{x}u)) \right) \right. \\
 & \left. \left| \left( y(v).q(b).(\bar{b} | t.(Q|\bar{q}f) | f.(\bar{q}f | \bar{y}v)) \right) \right) \right.
 \end{aligned} \tag{3.1}$$

On activation of the guard, a value is received through the query channel  $q$ , either a true value  $t$ , which triggers the execution of the body, or a false value  $f$ , which triggers the resending of the value on the input channel.

To extend this encoding to the full synchronous language is straightforward by dividing the send operation into two phases, a *query to send* and the actual transmission, see for instance the encoding used by Boudol (1992).

## 3.8 Time Durations and Parallelism

Once we have decided on a timed calculus without duration, we may want to re-introduce duration in some manner. We will now discuss three possible ways in which we can introduce durations in a calculus.

However, our choice is to not introduce duration at all in the actual calculus, but to observe the traces of the process to determine with which durations would be legal on a particular platform, something we develop in Chapter 5.

### 3.8.1 Adding a Duration

Some authors, for instance Yi (1991) and Chen (1992), construct a system with duration from a durationless systems by adding a delay  $\Delta_c$  to every atomic action, letting each step take  $c$  time units.

Unfortunately this is an unsound addition, both in our system and in the one developed by Yi (1991), because it leads to *maximum parallelism*, being equivalent to giving the system an infinite number of CPUs. The problem arises because the rule T-PAR (other calculi have a similar rule) allows us to perform two pieces of time progress in parallel. This is natural as long as time progress is simply processes waiting; on the other hand, if time progress represents work being performed, it means that two processes are allowed to work in parallel (and by extension, an infinite number). It is clear that durations cannot be allowed to run in parallel in general.

### 3.8.2 The Worker

One possible way to avoid the problem of parallelism would be to define a worker task and a single processor system, using the `now` construct that we introduced in Section 3.6:

$$\begin{aligned} \text{Worker} &\triangleq (\tilde{l}).l.\text{now} (\text{work}(p).\Delta_1.(\bar{p}|\tilde{l})) \mid \text{Worker}[l] \\ \text{SingleCPU} &\triangleq (\nu l)\tilde{l} \mid \text{Worker}[l] \end{aligned}$$

This enforces mutual exclusion over the  $\Delta_1$  portion, while suspending the process that should experience the delay. Note that one worker can be started for each processor in the system under discussion.

We can then introduce duration using the following macro:

$$\text{doWork} \triangleq (\nu c)\overline{\text{work}}\ c.c.$$

Note that in a calculus without the maximum progress property, we only get a lower bound on the duration of an operation, not the exact duration, since it does not stop us from taking more than one time step.

### 3.8.3 A Duration and Time Propagation

Another alternative is to add the duration to the reductions, and then to make a difference between work progress and time progress, which forces a slight complexity into the calculus. This is the path taken for instance by Berger with his  $\phi$ -function (Berger 2002), and by Timed CSP (Reed and Roscoe 1988) with the “system delay constant”. This induces a certain asymmetry in the rules so that time progresses in all parallel processes, but work progresses only in one of them. From our point of view this is undesirable since it stops us from performing classical sequential optimizations (Lemma 4.8), making it harder to generalize over different platform speeds.

## 3.9 Other Timed Process Calculi

There has been an almost uncountable number of suggestions for timed calculi, we will start by a brief taxonomy of the design of timed process calculi, and then look at two examples, based on CCS and the  $\pi$ -calculus respectively. More comprehensive surveys are given by Nicollin and Sifakis (1991) and Hennessy (1992). For an overview of the timed variants of CSP there is a brief but reference rich summary by Ouaknine and Schneider (2006).

It should also be noted that our combination of baseline and deadline is similar to the *integration* of  $ACP_\rho$  (Baeten and Bergstra 1991). In  $ACP_\rho$  actions are given independent timing, and the integration is seen as a (possibly infinite) composition over a time variable. Except for that similarity,  $ACP_\rho$  is a very different calculus from ours, in that it is more axiomatic and the operational semantics require a global time variable.

### 3.9.1 A Taxonomy of Timed Process Calculi

Depending on how we combine the answers to the questions in Section 3.1 we get calculi with widely varying properties. We will look at the impact some of these choices can have on the resulting calculus, in particular, we will consider the maximum progress property, duration and patience.

The choice regarding determinism is orthogonal to the other properties, so we will omit it in the following discussions. It mainly impacts the proofs and proof theories. When we talk about distributed systems it becomes more interesting again, but for regular systems, time determinism is the most common choice.

Furthermore, there is a wide range of options when it comes to choosing primitive operations, which would make this taxonomy needlessly complex. The primitive operation should of course be chosen depending on what the terms need to express.

Often, several of the choices are available within the same calculi, by using modifiers or several different primitives with different semantics. In particular, this is true for patience.

With this being said, we will now examine the eight possible combinations of answers to (Q2), (Q4), and (Q5) and the families of calculi they induce.

#### 3.9.1.1 With Maximum Progress Property

**Without duration.** These calculi will perform all actions as soon as they become available, and without any further time progress. This is a model that corresponds to the one we have about certain network protocols, where the processing inside the nodes is assumed to consume no time at all, and only delays *on the wire* are interesting. A calculus like this requires a mean to produce artificial delays.

**With neither duration nor patience.** A calculus like this would deadlock the moment an expected action is unavailable. It requires everything to be started, and finished, immediately. We know of no examples of this kind of calculus.

**Without duration but with patience.** This is the slightly more forgiving sibling to the previous calculus, it allows us to describe a reactive system where an event produces a process that cannot immediately synchronize, but which can remain there until a later event occurs. As an example we have TPL by Hennessy and Regan (1995) which was explicitly defined for tasks such as protocol verification.

**With duration.** This design space is filled with models trying to represent *real computers*, who react instantly on some external event, such as a timer event, and then run until they are finished.

**With duration but without patience.** This represents run-to-end semantics, where a process which does not finish is an error. As far as we know, there has never been a calculus of this type.

**With duration and with patience.** Having this combination of properties results in a calculus that is suitable for describing an actual system, with fixed durations for each step. An example of this kind of calculus is the work presented by Berger (2002) in his dissertation, see Section 3.9.2.

### 3.9.1.2 Without Maximum Progress Property

This family of calculi makes time progress optional, and makes the choice between time progress and work progress non-deterministic. It makes little sense to talk about such a calculus without patience, so we will assume patience.

**With duration.** Having a duration for untimed actions limit the amount of work that can be performed in each time interval, but without the MPP there is no actual guaranteed that it *is* performed. As far as we know there is no example of such a calculus.

**Without duration.** Without both duration and maximum progress everything is voluntary. Or in other words: We are at a maximum abstraction level, and we have to use the process trace to determine properties of terms.

This is the quadrant we have chosen, with the modification that we let patience (and thus also progress) be modified by our deadline argument. Another example is the calculus by Reed (1990).

### 3.9.2 A Timed $\pi$ -calculus: $\pi_t$

The first timed calculus we will look at is the timed  $\pi$ -calculus  $\pi_t$  developed by Berger (2002) (a briefer article by Berger (2004) is also available).

This calculus extends the  $\pi$ -calculus with a  $\mathbf{timer}^t(x(v).P, Q)$  construction that can either act as  $x(v).P$  if a message is received on  $x$  before  $t$  time units has passed, or as  $Q$  otherwise. Here  $Q$  is referred to as the time-out action.

The calculus has durations, using a time-passing function  $\phi(P)$  to pass time, so that if  $P \rightarrow P'$ , then  $P \mid Q \rightarrow P' \mid \phi(Q)$ .

The calculus does not have the maximum progress property, and it does have patience. This, together with the fact that is founded on the  $\pi$ -calculus makes it the closest relative to the work presented in this chapter. The main difference is that this calculus has durations, and the choice of timing primitive. Berger also extends the calculus in several interesting ways that we have not considered, in particular he focuses more on distributed systems.

### 3.9.3 Timed CCS

In his thesis Yi (1991) introduced Timed CSS as a timed extension to CCS, geared towards real-time systems. This calculus allows agents to observe the time at which a message is delivered, and has the primitive  $\epsilon(t)$  which denotes a delay of  $t$  time-units.

This calculus is durationless, patient and has the maximum progress property. It also has persistence and time determinism. The main difference to our calculus is that it merely observes time but does not fully react on it, Timed CSS also has maximum progress which we do not.

## 3.10 Discussion

An important property of our calculus is that we block time progress for processes which have run out of time, since the T-RUN rule requires the relative deadline to be greater than 0. It would be possible instead to adopt a *time-out* semantics where we introduce a rule that effectively terminated such processes:

$$\mathbf{before}_0 P \xrightarrow{\tau} \mathbf{0}.$$

This would allow us to express some things, such as timers, more easily but it would be a disadvantage when discussing scheduling of processes. Further, it would be slightly controversial to simply terminate processes that cross their deadline.



## Timed equivalencies

There are many approaches to defining equality between processes, where bisimilarities can be considered the standard representative for  $\pi$ -calculus.

Our intention here is not to delve too deeply into the world of bisimilarities, but rather to develop a foundation for reasoning about real-time programs. For a deeper look at bisimilarities see for instance book by Sangiorgi and Walker (2003). Sangiorgi's recent article on the history of bisimulation is extensive in its coverage of the development (Sangiorgi 2007).

We want our calculi to allow “optimizations”, that is, we want to allow us to replace *slow* terms with *faster* terms (or faster terms with slower terms, if we would so desire). We consider classical, sequential, calculation to be represented by steps of internal  $\tau$  actions, so to allow classical compiler optimizations is similar to allowing replacing  $\tau^n$  with  $\tau^m$  for all  $n$  and  $m$  (and this is indeed proven in Lemma 4.8). Of course, for some purposes, such as meeting a deadline on a specific platform, the number of actions taken does matter and this is what we explore in Chapter 5.

This makes us choose the (weak) bisimilarity as the basis for determining equality between process terms.

### 4.1 Definitions for Bisimilarities

First we define the weak transition relation, which is an  $\hat{A}$  *essential action* preceded or succeeded by any number (even zero) of non-essential, internal, actions.

**Definition 4.1** (Essential actions). *We let  $\gamma$  be the essential actions, and range over the set of non-internal actions,  $\gamma \in \Lambda \cup \{\delta\}$ .*

**Definition 4.2** (Weak transition relations). *We define the following syntax for weak transitions:*

1.  $\Longrightarrow$  is the reflexive and transitive closure of  $\xrightarrow{\tau}$ .
2.  $\xrightarrow{\gamma}$  is the composition  $\Longrightarrow \xrightarrow{\gamma} \Longrightarrow$ .

**Definition 4.3** ((Weak) Bisimulation). *A symmetric relation  $\mathcal{R}$  is a bisimulation if for all  $P \mathcal{R} Q$  we have*

1.  $P \xrightarrow{\gamma} P'$  implies  $Q \xrightarrow{\gamma} \mathcal{R} P'$ .
2.  $P \xrightarrow{\tau} P'$  implies  $Q \Longrightarrow \mathcal{R} P'$ .

We now define the actual bisimilarity. Note that it is sometimes referred to as weak bisimilarity, as opposed to strong bisimilarity, because it uses the weak transition relation. From now on, we will refer to it only as “bisimilarity”.

**Definition 4.4** (Bisimilarity). *Bisimilarity is the largest symmetric relation  $\approx$ , such that whenever  $P \approx Q$  we have*

1.  $P \xrightarrow{\gamma} P'$  implies  $Q \xrightarrow{\gamma} \approx P'$
2.  $P \xrightarrow{\tau} P'$  implies  $Q \Longrightarrow \approx P'$

*Note that this makes bisimilarity the union of all bisimulations.*

We refer to the subset of terms which do not contain our three timing constructions as *untimed* terms. For these terms we induce a sub-calculus using only the rules which do not mention timing constraints. This calculus is the untimed core of our timed calculus.

We have a similarly defined bisimilarity for this untimed core, which we will call  $\approx_u$ . The following two lemmas examine the relationship between timed and untimed processes.

**Lemma 4.5** (All untimed terms are patient). *Any term  $P$  without `beforet` or `aftert` in them can always make a  $\delta$ -transition without changing the term, so that  $P \xrightarrow{\delta} P$ .*

**Lemma 4.6** (All untimed bisimilarities are timed bisimilarities). *For every  $P$  and  $Q$ , if  $P \approx_u Q$  then  $P \approx Q$ .*

*Proof.* It suffices to show that  $\approx_u$  is a bisimulation in the sense of Definition 4.3.

Thus the induced untimed bisimulation  $\approx_u$  is also a timed bisimulation because for every  $P$  and  $Q$  such that  $P \approx_u Q$  we have

1.  $P \xrightarrow{\gamma} P'$  implies  $Q \xrightarrow{\gamma} \approx_u P'$ . By case analysis:

*Case  $\gamma = \delta$ :* In which case by Lemma 4.5 we have  $P = P'$  and  $Q \xrightarrow{\delta} Q'$ , thus  $Q' \approx_u P'$  by reflexivity.

*Case  $\gamma \neq \delta$ :* In which case the reduction must have used only rules from the core calculus, and thus it holds by the definition of  $\approx_u$ .

2.  $P \xrightarrow{\tau} P'$  implies  $Q \Rightarrow \approx_u P'$ . This holds by the definition of  $\approx_u$ .

□

## 4.2 Some Equalities

Equational reasoning is a fundamental and popular technique, it even turns up in introductory texts (O'Donnell et al. 2006), for reasoning about the properties of programs.

In timed process calculi it has been popular to give complete equational theories (Moller and Tofte 1990). We will settle for some useful equivalences, indeed congruences, which we expect from a real-time system:

**Theorem 4.7** (Some rules). *The following are bisimilar:*

1.  $\text{before}_t \text{after}_{t'} R \approx \text{after}_{t'} R$ .
2.  $\text{before}_t \text{before}_{t'} R \approx \text{before}_{\min(t,t')} R$ .
3.  $\text{after}_0 R \approx R$ .
4.  $\text{after}_t \text{after}_{t'} R \approx \text{after}_{t+t'} R$ .
5.  $\text{after}_t (R \mid S) \approx (\text{after}_t R) \mid (\text{after}_t S)$ .
6.  $\text{before}_t (R \mid S) \approx (\text{before}_t R) \mid (\text{before}_t S)$

*Proof.* See Appendix B.

□

It is also worth noting that the divergent process  $\perp$ , defined as

$$\perp \triangleq \tau.\perp, \tag{4.1}$$

is weakly bisimilar to  $\mathbf{0}$ . Note that this is not a congruence, because the terms  $\text{before}_t \mathbf{0}$  is not bisimilar to  $\text{before}_t \perp$ .

If we were to allow substitution of the inactive process for a divergent process the rule DONE may be applied not only to terminated processes, but also non-terminating processes. The rule then gets the following interpretation: *a process that can never perform any productive (non- $\tau$ ) action is done.*

### 4.3 The Admissibility of Optimizations

The classical compiler optimizations consist of transformations on sequential programs aiming to improve the performance of the program. The optimizations can be considered correct if the transformed program is observationally equivalent to the original one, that is if they produce the same result in all contexts.

An important question for a timed process calculus is how these sequential transformations can be allowed in the parallel setting, see for instance the example in the introductory section. For our purposes, a sequential optimization is one which affects only the number of internal  $\tau$ -steps of a given term. Giving duration to atomic actions prevents these optimizations, since it is then possible observe individual steps.

The following standard lemma (Sangiorgi and Walker 2003) shows that our calculus allows for this kind of optimization. This lemma will hold for most any calculus without durations for actions, since it is then in general impossible to build a process that detects  $\tau$ -steps.

**Lemma 4.8** ( $\tau$ -actions do not matter). *We can add or remove  $\tau$  actions as we desire, or more formally:*

$$\tau.P \approx P.$$

*Proof.* The union of the identity relation and the relation

$$\{(\tau.P, P) \mid P \text{ arbitrary}\}$$

is a bisimulation, because whenever  $P \xrightarrow{\alpha} P'$  by some rule, then  $\tau.P \xrightarrow{\tau} P \xrightarrow{\alpha} P'$  by TAU and the same rule. In the reverse, if  $\tau.P \xrightarrow{\tau} P$  by TAU, then  $P$  can take zero steps, and the bisimilarity holds.  $\square$

# Scheduling and timed processes

We gave some background to Scheduling in Section 2.2, and having developed our calculus we are now ready to move on to examining the scheduling of processes in our timed calculus. As we indicated in the introduction, we approach the scheduling problem through judgements on the traces of a program. We start out by defining what we mean by a feasible trace and a feasible program.

**Definition 5.1** (*N*-feasible trace). *A trace is N-feasible if it only contains  $\tau$  and  $\delta$  labels, and never has more than N consecutive  $\tau$ -labels.*

**Definition 5.2** (*N*-feasibility). *A program P is N-feasible if there exists an N-feasible trace of P that has an infinite length.*

Note that the definition is still useful for terminating programs, because the inactive process  $\mathbf{0}$  can perform arbitrary idling. We will equate the number of  $\delta$ -steps occurring to the left of a certain  $\tau$ -transition in the trace with the time of the transition. In the trace  $\{\tau, \delta, \tau, \delta, \tau\}$  the last  $\tau$ -transition happens after two time steps, or at time two.

The definition of an incorrect program is now straightforward:

**Definition 5.3** (Temporal Incorrectness). *A program is incorrect if it is not N-feasible for any finite number N.*

This notion of what it means for a program to be correct is suitable for uniprocessor systems, but does not directly carry over to the multi processor case. In the multi processor case we have to find a useful way to separate the traces from the individual processors before imposing our feasibility condition, this is a topic for future work.

We do not want to discuss optimizations and transformations in a platform specific manner, instead we will allow all transformations which do not introduce

incorrectness. For instance, it is apparent that the sequential optimizations we discussed in Section 4.3 do not affect the correctness of a program, only the platforms on which it is feasible.

In a realistic system we do not want a system to idle when it has work to perform, so we define what it means for a system be non-idling:

**Definition 5.4** (Non-idling). *A non-idling  $N$ -feasible trace is a trace such that there are exactly  $N$  consecutive  $\tau$ -labels between each  $\delta$ .*

## 5.1 Classical Scheduling Theory

Perhaps the most common form of feasibility analysis for real-time systems is scheduling analysis which allows the real-time system to be analyzed using relatively simple tools. The advantage of scheduling theory is that it is well understood in the real-time community, and a field of active research where many tools have been developed (Bouyssounouse and Sifakis 2005; Sha et al. 2004).

To apply classical scheduling theory to our calculus we start by showing that the Earliest Deadline First (EDF) algorithm as described by Liu and Layland (1973) carries over.

The first thing we need to do is to restrict the form our processes may take so that it corresponds to the original formulation. This is stated by Liu and Layland (1973) as the following five restrictions :

1. All tasks are periodic, with constant interval.
2. Deadlines are equal to the period.
3. All tasks are independent.
4. The run-time for each task is constant. We can relax this requirement, it is sufficient that the run-time has a fixed upper bound.

These requirements are translated into the following process calculus requirements on a periodic process:

**Definition 5.5** (Periodic Process). *A process  $P = \text{before}_D Q$  is a periodic process with period  $D$  and work load  $C$  if:*

1.  $\text{fn}(P) = \emptyset$ .
2. The body  $Q$  contains no  $\text{before}_{t'}$  with  $t' \neq D$ .

3. *There exists a sequence of arbitrarily interleaved  $\tau$  and  $\delta$  actions on  $P$  such that the number of  $\tau$  actions is bounded above by  $C$ , the number of  $\delta$  actions is exactly  $D$ , and the resulting process  $P'$  is also a periodic process with period  $D$  and work load  $C$ .*

The first condition enforces the independence restriction, while the second implies that the only active deadline constraint is the top level one ( $\mathbf{before}_D$ ). The final condition combines the requirements on periodicity and run-time. It is clear that a periodic process is also (at least)  $\lceil C/D \rceil$ -feasible, because the steps were arbitrarily interleaved. Observe that the work load  $C$  is not necessarily the execution time, but rather the number of work steps required.

**Example 5.6** (A periodic process). *The process  $P \triangleq \mathbf{before}_t (\tau.\tau.\tau.\tau.\mathbf{after}_t P)$  is a periodic process.*

*Intuitively, we can take five  $\tau$  steps, and then get rid of everything but  $\mathbf{after}_t P$ , and then take  $t$   $\delta$  steps. We are then back to  $P$ .*

Now, to be able to define the EDF-scheduling, we want a good definition of what we mean by the deadline of a system of processes; i.e., a generic process term:

**Definition 5.7** (Deadline of a process). *We define the earliest deadline of a process term to be the shortest exposed deadline:*

$$\begin{aligned}
 \mathbf{dl}(\mathbf{0}) &= \infty \\
 \mathbf{dl}(P \mid Q) &= \min(\mathbf{dl}(P), \mathbf{dl}(Q)) \\
 \mathbf{dl}(\nu z P) &= \mathbf{dl}(P) \\
 \mathbf{dl}(\pi.P) &= \infty \\
 \mathbf{dl}(\mathbf{before}_t P) &= \min(d, \mathbf{dl}(P)) \\
 \mathbf{dl}(\mathbf{after}_0 P) &= \mathbf{dl}(P) \\
 \mathbf{dl}(\mathbf{after}_t P), d \neq 0 &= \infty \\
 \mathbf{dl}(K[\tilde{a}]) &= \mathbf{dl}(P\{\tilde{a}/\tilde{x}\}) \quad \text{where } K \triangleq P[\tilde{x}] \\
 \mathbf{dl}(\mathbf{now} P) &= \infty
 \end{aligned}$$

*The last statement is of course only needed if we include  $\mathbf{now}$  in our calculus. A process  $P$  such that  $\mathbf{dl}(P) = \infty$  is deadline free.*

We want to reason about the scheduling of a system of periodic processes  $P_1 \mid \dots \mid P_n$ . Working under the independence assumption, we know that these top level processes will not interact with one another.

**Definition 5.8** (Parallel contexts). *We define the parallel process context  $\mathcal{C}$ :*

$$\mathcal{C} = [] \mid (\mathcal{C} \mid P) \mid (P \mid \mathcal{C})$$

*We also want to extend the deadline function to these contexts in the following fashion:*

$$\begin{aligned} \text{dl}([]) &= \infty \\ \text{dl}(P \mid \mathcal{C}) &= \min(\text{dl}(P), \text{dl}(\mathcal{C})) \\ \text{dl}(\mathcal{C} \mid P) &= \min(\text{dl}(\mathcal{C}), \text{dl}(P)) \end{aligned}$$

**Definition 5.9** (EDF Reduction). *We introduce a new symbol  $\tau_{\text{edf}}$  which is identical to a regular  $\tau$ -symbol, except for the extra label, and a special reduction rule which identifies an EDF reduction:*

$$\frac{P \xrightarrow{\tau} P' \quad \text{dl}(P) \leq \text{dl}(C)}{\mathcal{C}[P] \xrightarrow{\tau_{\text{edf}}} \mathcal{C}[P']} \text{EDF}$$

*We call a trace containing only  $\delta$  and  $\tau_{\text{edf}}$  steps an EDF-trace.*

**Lemma 5.10.** *If a process  $P$  has a  $N$ -feasible EDF-trace, then it has an  $N$ -feasible regular trace as well.*

**Theorem 5.11** (No Idling). *In a system that is not  $N$ -feasible there is a maximal  $l$  such that there is an  $N$ -feasible trace of length  $l$ , but not of length  $l + 1$ .*

*Then all  $N$ -feasible EDF traces of length  $l$  are non-idling (according to Definition 5.4).*

*Proof.* Proof by contradiction, similar to Liu and Layland (1973). See Appendix B for the full proof.  $\square$

**Theorem 5.12** (Utilization limit for EDF). *Assume that we have a system of periodic processes  $P_1 \mid \dots \mid P_m$ , where each process  $P_i$  has work load and relative deadline  $C_i$  and  $D_i$  respectively*

*Then the system has an infinite  $N$ -feasible EDF trace if and only if*

$$\sum_{i=0..n} \frac{C_i}{D_i} \leq N. \tag{5.1}$$

*Proof.* By an adaption of the proof by Liu and Layland (1973) to make judgements on the length of traces, and by use of Theorem 5.11.  $\square$

## 5.2 Other Process Calculi and Scheduling

Previous efforts at combining scheduling with timed process calculi have generally been more aimed at logical verification of timing properties than of adapting scheduling algorithms to the formalisms.

A representative example is ACSR-VP (Choi et al. 1995; Kwak et al. 1998). This process calculus has been developed for analyzing real-time systems. It uses logical methods such as bisimulation checking or constraint logic programming to resolve scheduling questions.

The ACSR-VP specification becomes similar to the logical, implicit time, formulation of Lamport (2005). This kind of formalism is more geared towards models and solving scheduling problems than it is meant for expressing programs.

Note that the trace-based approach to scheduling and real-time performance has some similarities to the notion of fairness between processes. Fairness implies that a process which wishes to progress eventually gets to progress, so that no one process is starved.

For instance, Costa and Stirling (1984) describe how to filter out the fair traces of CCS by modifying the semantics. Corradini et al. (2003) shows how fairness relates to timing in their CCS-like process calculus. While none of these deal directly with scheduling or meeting deadlines, we note that fairness can be seen as a first step towards scheduling since it is way to control the allowable traces of the system.



# Conclusions and Future work

## 6.1 Conclusions

We have created a new timed process calculus with several novel features. In particular we have eschewed durations to enable platform independent reasoning, and instead introduced baselines and deadlines into our calculus. To be able to reason about platform specific scheduling without giving a specified duration to actions we employ a new notion of  $N$ -feasibility.

The introduction of baselines and deadlines is something that has so far been done only to a limited degree in a rather more complex calculus (Baeten and Bergstra 1991), and never for a variant of the  $\pi$ -calculus.

After developing this calculus we have seen that it is possible to reason about the scheduling of terms, and restated some classical scheduling results as judgments on the traces of processes.

Our  $Ti\pi$ -calculus has proven to be both fairly expressive and easy to work with. The results shown in earlier chapters suggest that it will be a fruitful avenue of research for real-time systems. While there is a lot left to do, we have shown that it can be used to describe real-time systems, and reason about their timing properties.

## 6.2 Future work

A lot of possibilities reveal themselves once we have developed a timed process calculus. We will briefly discuss a few of them, roughly in order of decreasing interest.

**Timber integration.** While we draw inspiration from the Timber Programming language (Black et al. 2002), we have not done much to integrate the current work with it.

It would be suitable to look at what changes and extensions we need to be able to adapt the present formalism to the needs of a real programming language. This would include verifying that it has all the properties required to handle the Timber objects.

In particular, this would require us to handle stateful objects, and mutual exclusion between objects in a simple fashion. We also need to find a simple enough embedding of the functional parts of the Timber language.

**Parallel calculi.** Can we extend the calculus from a concurrent to a parallel calculus, i.e. one suitable for multi-CPU systems? The change should be reasonably straightforward to the calculus, but if we also want to retain some handle on scheduling it will probably be slightly more tasking.

The problem here is that the notion of  $N$ -feasibility does not immediately generalize to a parallel setting, because having two processors is not the same as having a single processor with double the performance, which would be the result of having the more traditional interleaved semantics. This means that we need to find a way to keep track of what is done on each processor, and capture that in our notion of traces.

**Modeling.** There has been a lot said about the problem of modeling real-time systems (Sifakis 2001; Henzinger and Sifakis 2006; Hessel and Pettersson 2006). We would like to explore the boundary between programming and modeling and see what the difference is between writing a program and creating a model.

Typically, modeling is considered to be more abstract than the implementation itself, there is the unspoken assumption that the model does not contain every aspect of the implementation. The question is how this relates to terms in the process calculus.

One interesting possibility is to find a link between our models and existing model checkers, in particular it would be interesting to see if a program or parts of a program could be directly checked in a model checker.

**Optimization.** If we accept the premise that the process calculus terms correspond to concurrent programs the same way lambda calculus does for functional programs, then it would make sense to look at optimization of terms in the process calculus. This should tie in nicely with the general trend towards automatic parallelization.

---

**More scheduling.** It should be possible to extend our coverage of scheduling algorithms to more interesting scheduling algorithms, for instance the SRP (Baker 1991) algorithm. This would make it possible to use the formalism for more realistic systems. Another alternative is to turn more towards logical verification of scheduling.

One central problem here is the question of shared locks/resources: Should they be explicitly introduced or coded implicitly in the calculus? And what kind of reasoning can we apply here to allow communication between top level processes in a more realistic fashion?



---

# APPENDIX A

---

## Glossary

This short glossary defines a few of the key terms used in this thesis. Most of them are defined more at length in other places, this is intended to be a short summary.

**Baseline.** The baseline of a task is the earliest time that execution of the task can be started.

**Büchi automaton.** A finite state automaton extended to infinite length inputs. Instead of requiring a single visit to an accepting state, it requires an infinite number of visits to an accepting state.

**Calculus of Communicating Systems (CCS)** A process calculi, the predecessor of the  $\pi$ -calculus. See Section 2.3.1 for an introduction.

**Critical section.** A section of a program to which *mutual exclusion* should apply, because it uses shared resources.

**Deadline.** The deadline of a task is the latest valid completion time for a task.

**Deadlock.** Two or more tasks under mutual exclusion are deadlocked if the tasks involved hold resources in such a way that none can progress without at least one of the others giving up its resource. See Example 2.3 for a clarification.

**Dense time.** Intuitively a time base  $\mathcal{T}$  is dense if it can always be divided into smaller pieces, or formally if

$$\forall t_1, t_2 \in \mathcal{T} : \exists t' \in \mathcal{T} : t_1 < t' < t_2.$$

See also *discrete time*.

**Determinism.** Things have only one way of happening. See also: *Time determinism*.

**Discrete time.** A time base that is not *dense* is discrete. See also *dense time*.

**Earliest Deadline First (EDF).** Scheduling algorithm in which processes are scheduled so that the process with the earliest deadline is given the highest priority. See: Section 2.2.1.

**Idling.** The act of existing without performing work. A process that is idling sees time pass without taking any actions.

**Insistent action.** *see Urgent actions*.

**Interleaving** description

**Laxity.** The laxity of a process describes how long the process is allowed to be blocked or idle to be able to meet its deadline. It is the difference between the relative deadline and the remaining work of the task.

**Labeled Transition System (LTS).** A general model for computation, given as a set of states and a labeled transition relation between states. Transitions are usually written  $s \xrightarrow{l} s'$ , interpreted as  $s$  goes to  $s'$  doing action  $l$ . First described by Keller (1976).

**Lock.** A lock is a variable that is shared between several processes so that it can be *held* only by one process at a time. This can serve to provide mutual exclusion for a critical section.

**Maximal parallelism.** The assumption that each process has its own processor available as needed, so that scheduling of processing resources is eliminated.

**Maximum Progress Principle (MPP).** Implies that idling is allowed only when no other, more productive, action is available. That is, a system must make as much progress on its work as possible.

**Mutual exclusion.** Protecting a resource so that only one process at a time may use it. See Section 2.1.1.

**Patience.** A term if patient if it is allowed to wait when it cannot perform work. Terms that are not patient are called *insistent*.

**Pi-calculus ( $\pi$ -calculus)** A process calculi, the successor of the CCS. See Section 3.2 for an introduction.

**Rate Monotonic (RM) Scheduling** The Rate Monotonic scheduling is a static priority scheduler where each task is given a priority that depending on its period. See Section 2.2.2.

**Slack.** See *Laxity*.

**Stack Resource Protocol (SRP).** The SRP is a protocol for scheduling tasks under mutual exclusion that is deadlock free and has guaranteed upper bounds. See also Section 2.2.3.

**Time dependency.** This is taken to mean that actions and time may depend on the actual time an event takes place. Chen (1992) considers this machine example:

$$M \stackrel{def}{=} \alpha(t)_0^\infty . \beta(s)_0^t . \text{NIL}.$$

Having time dependency requires processes to be able to observe time somehow.

**Time determinism.** A system is time deterministic if the progress of time is unambiguous. This means that there is only one possible way that a time interval can be passed.

This can be expressed like this (Nicollin and Sifakis 1991, Section 2.3.1):

$$\forall P, P', P'', d : P \xrightarrow{d} P' \wedge P \xrightarrow{d} P'' \Rightarrow P' \equiv P''$$

**Urgent action.** An action is urgent if idling is not permitted before the action takes place. If all actions are urgent the system obeys the *maximum progress principle*.

**Work-conserving.** A scheduler is said to be *work-conserving* if the system is idle only when there are no requests waiting to be served.

**Worst-Case Execution Time (WCET).** The WCET of a function is the upper bound on the execution time of the function. This value is required for most kinds of timing analysis. WCET analysis itself is undecidable in the general case, but the existence of WCET times is often assumed.

**Zeno machine.** Is a machine that performs infinite work in a finite (or even zero) time. The name comes from the greek philosopher Zeno, famous for his paradox about Achilles and the turtle.



---

# APPENDIX B

---

## Detailed Proofs

This chapter contains full proofs for some earlier theorems.

### B.1 Proofs About the Properties of the Calculus

**Lemma 3.4** (Time Confluence)

The  $Ti\pi$ -calculus without **now** is time-persistent; meaning that time progress can never remove an opportunity for a reduction step and reduction can never remove an opportunity for a time step.

This is equivalent to the following formal statement:

$$\forall R, R', S : R \xrightarrow{\alpha} R' \wedge R \xrightarrow{\delta} S \Rightarrow S \xrightarrow{\alpha} R' \wedge R' \xrightarrow{\delta} S.$$

Which can be visualized in the following commutative diagram:

$$\begin{array}{ccc} R & \xrightarrow{\delta} & S \\ \alpha \downarrow & & \downarrow \alpha \\ R' & \xrightarrow{\delta} & S' \end{array}$$

*Proof.* Remember that we know, by Lemma 3.5, that time progress is deterministic. In general we assume that time can progress in  $R$ , otherwise the lemma holds vacuously.

We proceed by induction on the structure of  $R$ :

*Case 0:* Can take no actions, the theorem holds vacuously.

*Case  $P \mid Q$ :* By T-PAR we get  $S = P' \mid Q'$  where  $P \xrightarrow{\delta} P'$  and  $Q \xrightarrow{\delta} Q'$ . There are three possible ways to take actions (and their symmetries):

*Case PAR-L:*  $P \mid Q \xrightarrow{\alpha} P' \mid Q = R'$ , because  $P$  can take the action  $\alpha$ , and  $\text{bn}(\alpha) \cap \text{fn}(Q) = \emptyset$ . Further,  $P' \mid Q \xrightarrow{\delta} P''' \mid Q'$ , where  $P' \xrightarrow{\delta} P''$  and  $Q \xrightarrow{\delta} Q'$ .

By induction we know that if  $P \xrightarrow{\delta} P'$ ,  $P \xrightarrow{\alpha} P'$  and  $P' \xrightarrow{\delta} P''$  then  $P' \xrightarrow{\alpha} P'''$ , thus  $P' \mid Q' \xrightarrow{\alpha} P''' \mid Q'$  by PAR-L, and we know that  $\text{fn}(Q) = \text{fn}(Q')$  by Fact 3.6.

*Case COMM-L:*  $P \mid Q \xrightarrow{\tau} P'' \mid Q'' = R'$ , since  $P \xrightarrow{\bar{x}y} P''$  and  $Q \xrightarrow{xy} Q''$ . Further,  $P'' \mid Q'' \xrightarrow{\delta} P''' \mid Q'''$ , where  $P'' \xrightarrow{\delta} P'''$  and  $Q'' \xrightarrow{\delta} Q'''$ .

By induction we know that if  $P \xrightarrow{\delta} P'$ ,  $P \xrightarrow{\bar{x}y} P''$  and  $P'' \xrightarrow{\delta} P'''$  then  $P' \xrightarrow{\bar{x}y} P'''$  and by the same argument  $Q' \xrightarrow{xy} Q'''$ . Thus  $P' \mid Q' \xrightarrow{\tau} P''' \mid Q'''$  by COMM-L.

*Case CLOSE-L:*  $P \mid Q \xrightarrow{\tau} (\nu z) P'' \mid Q'' = R'$ , since  $P \xrightarrow{\bar{x}(y)} P''$ ,  $Q \xrightarrow{xy} Q''$  and  $z \notin \text{fn}(Q)$ . Further,  $P'' \mid Q'' \xrightarrow{\delta} P''' \mid Q'''$ , where  $P'' \xrightarrow{\delta} P'''$  and  $Q'' \xrightarrow{\delta} Q'''$ .

By induction we know that if  $P \xrightarrow{\delta} P'$ ,  $P \xrightarrow{\bar{x}(y)} P''$  and  $P'' \xrightarrow{\delta} P'''$  then  $P' \xrightarrow{\bar{x}(y)} P'''$  and by the same argument  $Q' \xrightarrow{xy} Q'''$ . Thus  $P' \mid Q' \xrightarrow{\tau} (\nu z) P''' \mid Q'''$  by CLOSE-L, we know that  $\text{fn}(Q) = \text{fn}(Q')$  by Fact 3.6 so  $z \notin \text{fn}(Q')$ .

*Case TIMECOMM:* For this case we have  $R = \text{before}_t P \mid \text{before}_{t'} Q$  and  $S = \text{before}_{t-1} P' \mid \text{before}_{t'-1} Q'$ , where  $P \xrightarrow{\delta} P'$  and  $Q \xrightarrow{\delta} Q'$ .

We get  $R' = \text{before}_t P'' \mid \text{before}_{t'} Q''$  by TIMECOMM performing action  $\alpha$  assuming  $P \mid Q \xrightarrow{\alpha} P'' \mid Q''$ . And  $S' = \text{before}_{t-1} P''' \mid \text{before}_{t'-1} Q'''$ , if  $P'' \xrightarrow{\delta} P'''$  and  $Q'' \xrightarrow{\delta} Q'''$ .

We know that  $P \mid Q \xrightarrow{\delta} P' \mid Q'$ ,  $P \mid Q \xrightarrow{\alpha} P'' \mid Q''$  and  $P'' \mid Q'' \xrightarrow{\delta} P''' \mid Q'''$ , then by induction we know that  $P' \mid Q' \xrightarrow{\alpha} P''' \mid Q'''$ .

So finally we have that

$$\text{before}_{t-1} P' \mid \text{before}_{t'-1} Q' \xrightarrow{\alpha} \text{before}_{t-1} P''' \mid \text{before}_{t'-1} Q''',$$

by TIMECOMM.

*Case  $(\nu z) P$ :* Gives us  $S = (\nu z) P'$  where  $P \xrightarrow{\delta} P'$ .

There are two possible rules by which we can deduce  $R'$

*Case RES:*  $R' = (\nu z) P''$  when  $P \xrightarrow{\alpha} P''$  and  $z \notin n(\alpha)$ . We get  $(\nu z) P'' \xrightarrow{\delta} (\nu z) P'''$  when  $P'' \xrightarrow{\delta} P'''$ .

By induction we know that if  $P \xrightarrow{\delta} P'$ ,  $P \xrightarrow{\alpha} P''$  and  $P'' \xrightarrow{\delta} P'''$  then  $P' \xrightarrow{\alpha} P'''$ . Therefore  $(\nu z) P' \xrightarrow{\alpha} (\nu z) P'''$  by RES.

*Case OPEN:* Exactly as for RES.

*Case  $\pi.P$ :* By T-SEQ  $S = \pi.P'$ , where  $P \xrightarrow{\delta} P'$ . We have three different cases for  $\pi$ :

*Case  $\bar{x}y$ :*  $R' = P$  by OUT, while taking action  $\bar{x}y$ , thus  $S' = P'$ . Finally  $\bar{x}y.P' \xrightarrow{\bar{x}y} P'$  by OUT.

*Case  $x(z)$ :*  $R' = P$  by INP, while performing action  $x(z)$ , thus  $S' = P'$ . Finally  $x(z).P' \xrightarrow{xy} P'$  by INP.

*Case  $\tau$ :*  $R' = P$  by TAU, while performing action  $\tau$ , thus  $S' = P'$ . Finally  $\tau.P' \xrightarrow{\tau} P'$  by TAU.

*Case  $\text{before}_t P$ :* If  $t = 0$  there is no valid time step, and the lemma hold vacuously. Let us assume  $t > 0$ , then  $S = \text{before}_{t-1} P$ ,  $P \xrightarrow{\delta} P'$ .

We have two possible reduction rules:

*Case TIMERES:* We get  $R' = \text{before}_t P''$ , when  $P \xrightarrow{\alpha} P''$ . Further, we get  $\text{before}_t P'' \xrightarrow{\delta} \text{before}_{t-1} P'''$  when  $P'' \xrightarrow{\delta} P'''$ .

By induction we know that if  $P \xrightarrow{\delta} P'$ ,  $P \xrightarrow{\alpha} P''$  and  $P'' \xrightarrow{\delta} P'''$  then  $P' \xrightarrow{\alpha} P'''$ . Therefore  $\text{before}_{t-1} P' \xrightarrow{\alpha} \text{before}_{t-1} P'''$  by TIMERES.

*Case BREAKFREE:* In this case  $R = \text{before}_t \text{after}_{t'} P$  and  $R' = \text{after}_{t'} P$  by BREAKFREE.

There are two cases for  $\text{after}_{t'}$  which give different deductions:

*Case  $t' = 0$ :*  $S = \text{before}_{t-1} \text{after}_0 P'$ , where  $P \xrightarrow{\delta} P'$ . And  $\text{after}_0 P \xrightarrow{\delta} \text{after}_0 P'$ .

Finally  $\text{before}_{t-1} \text{after}_0 P' \xrightarrow{\tau} \text{after}_0 P'$  using BREAKFREE.

*Case  $t' > 0$ :*  $S = \text{before}_{t-1} \text{after}_{t'-1} P$  and  $\text{after}_{t'} P \xrightarrow{\delta} \text{after}_{t'-1} P$ .

Finally  $\text{before}_{t-1} \text{after}_{t'-1} P' \xrightarrow{\tau} \text{after}_{t'-1} P$ , using BREAKFREE.

*Case  $\text{after}_t P$ :* If  $t > 0$  then  $R$  can perform no action and the theorem is satisfied vacuously, we assume  $t = 0$  in the following.

The only action  $R$  can take is DROPAFTER, emitting  $\tau$  to get  $R' = P$ . By T-READY we get  $S = \text{after}_0 P'$ , where  $P \xrightarrow{\delta} P'$ .

Finally we have  $\text{after}_0 P' \xrightarrow{\tau} P'$  by DROPAFTER.

*Case  $K[\tilde{a}]$ :* We let  $Q = P\{\tilde{a}/\tilde{x}\}$ , where  $K \triangleq P[\tilde{x}]$ .

We have one possible action, CONST, emitting  $\alpha$  to get  $R' = Q'$  from  $Q \xrightarrow{\alpha} Q'$ .

And  $Q' \xrightarrow{\delta} Q'''$  for some  $Q'''$ . Using T-CONST we get  $S = Q''$  where  $Q \xrightarrow{\delta} Q''$ .

By induction, since  $Q \xrightarrow{\alpha} Q'$ ,  $Q' \xrightarrow{\delta} Q'''$  and  $Q \xrightarrow{\delta} Q''$  we know that  $Q'' \xrightarrow{\delta} Q'''$ .

□

**Lemma 3.9** (Time Persistence) The  $Ti\pi$ -calculus with **now** is time-persistent; meaning that time progress can never remove an opportunity for a reduction step.

This is equivalent to the following statement:

$$\forall R, R', S : R \xrightarrow{\alpha} R' \wedge R \xrightarrow{\delta} S \Rightarrow \exists S'' : S \xrightarrow{\alpha} S''$$

*Proof.* Remember that we know, by Lemma 3.5, that time progress is deterministic. In general we assume that time can progress in  $R$ , otherwise the lemma holds vacuously.

We proceed by induction on the structure of  $R$ :

*Case 0:* Can take no actions, the theorem holds vacuously.

*Case  $P \mid Q$ :* We get  $S = P' \mid Q'$  where  $P \xrightarrow{\delta} P'$  and  $Q \xrightarrow{\delta} Q'$ .

There are three possible ways to take actions (and their symmetries):

*Case PAR-L:* We can perform the action  $\alpha$  because  $P$  can take the action  $\alpha$ , and because  $\text{bn}(\alpha) \cap \text{fn}(Q) = \emptyset$ . By induction we know that  $P'$  can still take action  $\alpha$  and by Fact 3.6 we have  $\text{fn}(Q') = \text{fn}(Q)$ , thus  $P' \mid Q'$  can still perform  $\alpha$ .

*Case COMM-L:* By induction  $P'$  can still perform  $\bar{x}y$  and  $Q'$  can perform  $xy$ , thus  $P' \mid Q'$  can take action  $\tau$ .

*Case CLOSE-L:* Similarly, by induction  $P'$  can still take action  $\bar{x}(z)$ ,  $Q'$  can take  $xy$  and  $\text{fn}(Q) = \text{fn}(Q')$ . Hence,  $P' \mid Q'$  can take action  $\tau$ .

*Case  $(\nu z)P$ :* Gives us  $S = (\nu z)P'$  where  $P \xrightarrow{\delta} P'$ . By RES  $R$  can perform action  $\alpha$  if  $P$  can perform  $\alpha$ , by induction  $P'$  can then take action  $\alpha$  and so can  $S$ .

*Case  $\pi.P$ :* By T-SEQ  $S = \pi.P'$ ,  $P \xrightarrow{\delta} P'$ . We have three different cases for  $\pi$ :

*Case  $\bar{x}y$ :* Can perform  $\bar{x}y$  by OUT. And since  $S = \bar{x}y.P'$ , it can perform  $\bar{x}y$  by OUT.

*Case  $x(z)$ :* Can perform  $x(z)$  by INP. And since  $S = \bar{x}y.P'$ , it can perform  $x(z)$  by INP.

*Case  $\tau$ :* Can perform  $\tau$  by TAU. And since  $S = \tau.P'$ , it can perform  $\tau$  by TAU.

*Case  $\text{before}_d P$ :* If  $d = 0$  there is no valid time step, and the lemma hold vacuously. Let us assume  $d > 0$ , then  $S = \text{before}_{d-1} P'$ ,  $P \xrightarrow{\delta} P'$ .

We have two possible reduction rules:

*Case TIMERES:* Since  $\mathbf{before}_d P$  can take action  $\alpha$  if  $P$  can take action  $\alpha$ , and by induction  $P'$  can then perform  $\alpha$  as well. Consequently  $\mathbf{before}_{d-1} P'$  can perform  $\alpha$ .

*Case BREAKFREE:* In this case we will have  $R = \mathbf{before}_d \mathbf{after}_b P$ . Then  $S = \mathbf{before}_{d-1} \mathbf{after}_{b-1} P$ , to which BREAKFREE also applies. So  $S$  can perform  $\tau$  if  $R$  could.

*Case after<sub>b</sub> P:* If  $b > 0$  then  $R$  can perform no action and the theorem is satisfied vacuously, we assume  $b = 0$  in the following.

By RESAFTER  $R$  can perform  $\alpha$  if  $P$  can. By induction then.

*Case now P:* Gives  $S = \mathbf{now} P$ , since  $S$  and  $R$  are structurally equivalent it is clear that they can take the same set of actions. In fact this is just the  $\tau$  action of START.

*Case  $K[\tilde{a}]$ :* We let  $Q = P\{\tilde{a}/\tilde{x}\}$ , where  $K \triangleq P[\tilde{x}]$ . We have one possible action, CONST, emitting  $\alpha$  to get  $R' = Q'$  from  $Q \xrightarrow{\alpha} Q'$ . Using T-CONST we get  $S = Q''$  where  $Q \xrightarrow{\delta} Q''$ . By induction,  $Q''$  can take the action  $\alpha$  since  $Q$  could.

□

## B.2 Bisimilarities Proofs

Proofs to Chapter 4.

**Theorem 4.7** (Some Rules) The following are bisimilar:

1.  $\mathbf{before}_t \mathbf{after}_{t'} R \approx \mathbf{after}_{t'} R$ .
2.  $\mathbf{before}_t \mathbf{before}_{t'} R \approx \mathbf{before}_{\min(t,t')} R$ .
3.  $\mathbf{after}_0 R \approx R$ .
4.  $\mathbf{after}_t \mathbf{after}_{t'} R \approx \mathbf{after}_{t+t'} R$ .
5.  $\mathbf{after}_t (R \mid S) \approx (\mathbf{after}_t R) \mid (\mathbf{after}_t S)$ .
6.  $\mathbf{before}_t (R \mid S) \approx (\mathbf{before}_t R) \mid (\mathbf{before}_t S)$

*Proof.* We use the fact that time progress is deterministic. That it is also a congruence follows from the fact that none of the timed constructs contain names and that we preserve  $R$  and  $S$ .

1. We have  $P = \mathbf{before}_t \mathbf{after}_{t'} R$  and  $Q = \mathbf{after}_{t'} R$ . But since we also have  $P = \mathbf{before}_t \mathbf{after}_{t'} R \xrightarrow{\tau} \mathbf{after}_{t'} R = Q$  (by BREAKFREEE) we know that  $Q$  simulates  $P$ .

For the converse we have the following cases that may apply to  $Q$ :

*Case  $\alpha$ -transition:* In this case  $P$  can always take the same step, since it must always originate with  $R$ . Since only  $R$  changes structure the resulting terms are bisimilar.

*Case  $\tau$ -transition:* Is either BREAKFREE, in which case  $P$  takes no action and the terms are now structurally equivalent, or something originating with  $R$  in which case  $P$  takes the same transition and they remain bisimilar.

*Case  $\delta$ -transition:*  $\mathbf{before}_t \mathbf{after}_{t'} R \xrightarrow{\delta} \mathbf{before}_{t-1} \mathbf{after}_{t'-1} R'$  only if  $\mathbf{after}_t R \xrightarrow{\delta} \mathbf{after}_{t-1} R'$ , by RESBEFORE, so the resulting terms are bisimilar. (Similarly for  $t = 1$ ).

2. For cases where  $R$  is on the form  $\mathbf{after}_t R'$  we can apply equality (1), above, to both sides. We will assume that  $R$  is not on that form in the following. Similarly for the case where  $R = \mathbf{0}$ .

In the case where the action is a regular  $\tau$  or  $\alpha$  action we know that it only affects  $R$ , since it passes through  $\mathbf{before}_t$  using the RESBEFORE rule.

We have  $P = \mathbf{before}_t \mathbf{before}_{t'} R$  and  $\mathbf{before}_{\min(t,t')} R$ . We proceed by induction on  $\min(t, t')$ .

We now look at the timed reductions:

*Case  $\min(t, t') = 0$ :* In this case the right side cannot take timed actions, because  $\mathbf{before}_0 R$  can take no timed action. The left side cannot take a timed action either, because either  $t = 0$  or  $t' = 0$ . So it holds vacuously.

*Case  $\min(t, t') > 0$ :* In this case neither  $t$  nor  $t'$  are 0, so we can take the step  $\mathbf{before}_t \mathbf{before}_{t'} R \xrightarrow{\delta} \mathbf{before}_{t-1} \mathbf{before}_{t'-1} R$ , and also  $\mathbf{before}_{\min(t,t')} R \xrightarrow{\delta} \mathbf{before}_{\min(t,t')-1} R$ . Note that  $\min(t-1, t'-1) = \min(t, t') - 1$ . So by induction the theorem holds.

3. We have  $P = \mathbf{after}_0 R$  and  $Q = R$ .  
The only possible  $\alpha$  action is  $\mathbf{after}_0 R \xrightarrow{\alpha} \mathbf{after}_0 R'$ , by using RESAFTER and assuming that  $R \xrightarrow{\alpha} R'$ , and similarly using T-READY for  $\delta$  actions.
4. We have  $P = \mathbf{after}_t \mathbf{after}_{t'} R$  and  $Q = \mathbf{after}_{t+t'} R$ .

We proceed by induction on  $t + t'$ .

*Case  $t + t' = 0$ :* In the base case, either  $t = 0$  or  $t' = 0$ , if for instance  $t'$  is zero, then we can use the fact that  $\mathbf{after}_0 R \approx R$  to rewrite the relation as  $\mathbf{after}_t R \approx \mathbf{after}_{t+t'} R$ , which is trivially true. Similarly if  $t = 0$ .

*Case  $t + t' \neq 0$ :* We assume that  $\mathbf{after}_{t-1} \mathbf{after}_{t'} R \approx \mathbf{after}_{t+t'-1} R$ .

The left hand side cannot take an  $\alpha$  action, because either  $b \neq 0$  or  $t' \neq 0$ , and thus we cannot use RESAFTER. Similarly for the right hand side, knowing that  $t + t' \neq 0$ .

Taking a  $\delta$  action we have  $\mathbf{after}_t \mathbf{after}_{t'} R \xrightarrow{\delta} \mathbf{after}_{t-1} \mathbf{after}_{t'} R$ , using T-WAIT, while  $\mathbf{after}_{t+t'} R \xrightarrow{\delta} \mathbf{after}_{t+t'-1} R$ , using T-WAIT. So by the induction assumption the relation holds.

5. We have  $P = \mathbf{after}_t (R \mid S)$  and  $Q = (\mathbf{after}_t R) \mid (\mathbf{after}_t S)$ .

We proceed by induction on  $t$ .

*Case  $t = 0$ :* If  $t = 0$ , then  $\mathbf{after}_0 (R \mid S) \approx R \mid S$ , by equality (3), and similarly  $(\mathbf{after}_0 R) \mid (\mathbf{after}_0 S) \approx R \mid S$ . So the relation holds.

*Case  $t \neq 0$ :* Assume that the relation holds for  $t - 1$ .

We cannot take an  $\alpha$  action, because the rule RESAFTER can only be used if  $t = 0$ .

If we take a  $\delta$  action, then  $\mathbf{after}_t (R \mid S) \xrightarrow{\delta} \mathbf{after}_{t-1} (R \mid S)$  by T-WAIT. And  $(\mathbf{after}_t R) \mid (\mathbf{after}_t S) \xrightarrow{s} (\mathbf{after}_t R - 1) \mid (\mathbf{after}_{t-1} S)$  by T-PAR and T-WAIT. And by the induction assumption we know that  $(\mathbf{after}_t R - 1) \mid (\mathbf{after}_{t-1} S) \approx \mathbf{after}_{t-1} (R \mid S)$ .

6. We have  $P = \mathbf{before}_t (R \mid S)$  and  $Q = (\mathbf{before}_t R) \mid (\mathbf{before}_t S)$ . We proceed by case analysis on the deadline  $d$ :

*Case  $d = 0$ :* Then no further time progress is possible in either case.

We have two sub-cases:

*Case  $R$  and  $S$  do not communicate:* If  $R \xrightarrow{\alpha} R'$  by some rule we have  $\mathbf{before}_t (R \mid S) \xrightarrow{\alpha} \mathbf{before}_t (R' \mid S)$  by the same rule and PAR-L and RESBEFORE. But then  $(\mathbf{before}_t R) \mid (\mathbf{before}_t S)$  by the same rule and RESBEFORE and PAR-L. The symmetric cases are trivial.

*Case  $R$  and  $S$  communicate:* Assume that  $R \mid S \xrightarrow{\tau} R' \mid S'$ , we know it must do so by either COMM-L or CLOSE-L, we look at the two cases:

*Case : COMM-L* It  $R \xrightarrow{\bar{x}y} R'$  and  $S \xrightarrow{x(y)} S'$  then by COMM-L and RESBEFORE we get  $\mathbf{before}_t (R \mid S) \xrightarrow{\tau} \mathbf{before}_t (R' \mid S')$ .

We will then have  $\mathbf{before}_t R \xrightarrow{\bar{x}y} \mathbf{before}_t R'$  and  $\mathbf{before}_t S \xrightarrow{x(y)} \mathbf{before}_t S'$  by the original rules and RESBEFORE.

Thus  $(\mathbf{before}_t R) \mid (\mathbf{before}_t S) \xrightarrow{\tau} (\mathbf{before}_t R') \mid (\mathbf{before}_t S')$  by the original rules, RESBEFORE and COMM-L.

Similar for the symmetric cases.

*Case* : CLOSE-L It  $R \xrightarrow{\bar{x}(y)} R'$  and  $S \xrightarrow{x(y)} S'$  then by CLOSE-L and RESBEFORE we get  $\text{before}_t (R \mid S) \xrightarrow{\tau} \text{before}_t (\nu z) (R' \mid S')$ .

We will then have  $\text{before}_t R \xrightarrow{\bar{x}(y)} \text{before}_t R'$  and  $\text{before}_t S \xrightarrow{xy} \text{before}_t S'$  by the original rules and RESBEFORE.

Thus  $(\text{before}_t R) \mid (\text{before}_t S) \xrightarrow{\tau} (\nu z) (\text{before}_t R') \mid (\text{before}_t S')$  by the original rules, RESBEFORE and CLOSE-L.

*Case*  $d \neq 0$ : We will only examine the time progress case, the other cases are identical.

If  $R \xrightarrow{\delta} R'$  and  $S \xrightarrow{\delta} S'$ , then  $\text{before}_t (R \mid S) \xrightarrow{\delta} \text{before}_{t-1} (R' \mid S')$ , with T-RUN, T-PAR and the original rules.

Similarly  $(\text{before}_t R) \mid (\text{before}_t S) \xrightarrow{\delta} (\text{before}_{t-1} R') \mid (\text{before}_{t-1} S')$  using T-PAR, T-RUN and the original rules.

□

## B.3 Scheduling Proofs

Proofs for Chapter 5.

**Theorem 5.11** (No Idling) In a system that is not  $N$ -feasible there is a maximal  $l$  such that there is an  $N$ -feasible trace of length  $l$ , but not of length  $l + 1$ .

Then all  $N$ -feasible EDF traces of length  $l$  are non-idling (according to Definition 5.4).

*Proof.* Adapted from Liu and Layland (1973), proof by contradiction.

Suppose that there is an  $N$ -feasible EDF trace of length  $l$  and that it contains idling. Let  $i$  denote the index of the first position in the trace such that the suffix after  $i$  is non-idling.

Assume that the first process released after  $i$  is released  $d$  time units after  $i$ . We can then release this process  $d$  time units early, at  $i$ . Since there was no idle between  $i$  and  $t$ , there will be no idle time even after the task is moved up. Moreover, we know that it is still not possible to make an  $N$ -feasible trace longer than  $t$ , so the new system must have an  $N$ -feasible trace equal to or shorter than  $t$ .

This reasoning can be repeated for all tasks, thus we conclude that if all tasks were started at  $i$  there would be a system without an idle period, but still with the same maximal trace. □

**Theorem 5.12** (Utilization limit for EDF) Assume that we a system of periodic processes  $P_1 \mid \dots \mid P_m$ , where each process  $P_i$  has running time and relative deadline  $C_i$  and  $D_i$  respectively

Then the system has an infinite  $N$ -feasible EDF trace if and only if

$$\sum_{i=0..n} \frac{C_i}{D_i} \leq N. \tag{B.1}$$

*Proof.* This proof is directly adapted from Liu and Layland (1973), the changes are mainly to make it apply to our underlying formalism.

To show necessity, we observe that the total number of  $\tau$ -steps required between the beginning of the trace and the  $D_1 D_2 \dots D_n$ :th  $\delta$ -step is

$$\begin{aligned} & (D_2 D_3 \dots D_n) C_1 + (D_1 D_3 \dots D_n) C_2 + \\ & \dots + (D_1 D_2 \dots D_{n-1}) C_n. \end{aligned}$$

It is clear that the trace can only contain  $N \times D_1 D_2 \dots D_n$   $\tau$ -steps up to this point. So, because

$$\begin{aligned} & (D_2 D_3 \dots D_n) C_1 + (D_1 D_3 \dots D_n) C_2 + \\ & \dots + (D_1 D_2 \dots D_{n-1}) C_n > N \times (D_1 D_2 \dots D_n) \end{aligned}$$

is equal to the statement

$$C_1/D_1 + C_2/D_2 + \dots C_n/D_n > N,$$

we can conclude that there can be no valid trace for these systems.

The following proof by contradiction establishes *sufficiency*. Assume that the condition in Equation B.1 holds, and yet there is no infinite  $N$ -feasible EDF trace.

Since the processes are periodic we know from Definition 5.5 that they have a valid sequence of actions in isolation, the requirement that the interlacing be arbitrary implies that the  $\delta$ -steps have no impact on the progress of the term itself. Thus, if there is no valid trace it is because we run into a deadline forcing us to take more than  $N$  consecutive  $\tau$ -steps in between two  $\delta$  steps.

In this case there *is* a maximal (but finite)  $N$ -feasible trace with length  $l$ , such that  $l < tN$  for some time  $t$ . That is, the overflow appears at time  $t$ . And we know by Theorem 5.11 that this trace is non-idling.

Since we have an overrun at time  $t$  we have at least one process such that its relative deadline is 0 at this point. Let  $P_{n_1}, P_{n_2}, \dots$  denote the processes with non-zero deadline.

We examine two different cases for the processes with non-zero deadline:

*Case* none of the processes have had any work carried out before  $t$ : The total demand on  $\tau$  steps generated by the system up until  $t$  is

$$\lfloor t/D_1 \rfloor C_1 + \lfloor t/D_2 \rfloor C_2 + \dots + \lfloor t/D_m \rfloor C_m.$$

Since there is no idle time, and we have an overflow we know that we must have:

$$\lfloor t/D_1 \rfloor C_1 + \lfloor t/D_2 \rfloor C_2 + \dots + \lfloor t/D_m \rfloor C_m > Nt.$$

But, since  $\lfloor x \rfloor \leq x$  for all  $x$ :

$$(t/D_1)C_1 + (t/D_2)C_2 + \dots + (t/D_m)C_m > Nt$$

and

$$(C_1/D_1) + (C_2/D_2) + \dots + (C_m/D_m) > N,$$

which is a contradiction of Equation B.1.

*Case* at least one of the processes have had work carried out before  $t$ : We know that we have an overflow at  $t$ , so there must be a point  $t'$ ,  $t' < t$  such that none of the processes  $P_{n_1}, P_{n_2}, \dots$  had work carried out in that interval. In other words, in the interval  $[t' \dots t]$  only processes with deadlines at or before  $t$  will be executed. Further, the fact that these processes are processed before  $t'$  implies that all requests initiated before  $t'$  and with deadlines at or before  $t$  have finished before  $t'$  (since otherwise processes with a later deadline would not be allowed to execute). Thus, the total demand of processor time between  $t'$  and  $t$  is less than or equal to:

$$\lfloor (t - t')/D_1 \rfloor C_1 + \lfloor (t - t')/D_2 \rfloor C_2 + \dots + \lfloor (t - t')/D_m \rfloor C_m.$$

But, the fact that we have an overflow implies that

$$\lfloor (t - t')/D_1 \rfloor C_1 + \lfloor (t - t')/D_2 \rfloor C_2 + \dots + \lfloor (t - t')/D_m \rfloor C_m > N(t - t'),$$

which again implies that

$$(C_1/D_1) + (C_2/D_2) + \dots + (C_m/D_m) > N,$$

which is again a contradiction of Equation B.1. Thus the theorem holds. □

---

## REFERENCES

---

- Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
- Jos C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005. ISSN 0304-3975. doi: <http://dx.doi.org/10.1016/j.tcs.2004.07.036>.
- Jos C. M. Baeten and Jan A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(3):142–188, 1991.
- Theodore. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991. ISSN 0922-6443.
- Henk P. Barendregt. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, March 2000. URL <ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf>.
- Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.
- Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.
- Martin Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, Dept. of Computing, 2002.
- Martin Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous ir-Calculi. *CONCUR 2004—concurrency Theory: 15th International Conference, London, UK, 2004*.
- Arthur Bernstein and Jr Paul K. Harter. Proving real time properties of programs with temporal logic. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, Operating Systems Review 15, 5, pages 1–11, New York, 1981. ACM.

- Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon Health & Science University, April 2002.
- Greg Bollella and James Gosling. The Real-Time Specification for Java. *Computer*, pages 47–54, 2000.
- Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.
- Gerard Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, INRIA, 1992.
- Bruno Bouyssounouse and Joseph Sifakis, editors. *The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer, 2005.
- Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3rd edition, 2001.
- Giorgio C. Buttazzo. Rate monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29:5–26, 2005.
- Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. *The Semantic Layers of Timber*, volume 2895. Springer Berlin, January 2003.
- Liang Chen. *Timed Processes: Models, Axioms and Decidability*. PhD thesis, University of Edinburgh, 1992.
- Jin-Young Choi, Insup Lee, and Hong-Liang Xie. The Specification and Schedulability Analysis of Real-Time Systems using ACSR. *IEEE Real-Time Systems Symposium*, pages 266–275, 1995.
- Flavio Corradini, Maria Rita Di Berardini, and Walter Vogler. Relating Fairness and Timing in Process Algebras. *Lecture Notes in Computer Science*, pages 446–460, 2003.
- Gerardo Costa and Colin Stirling. A fair calculus of communicating systems. *Acta Informatica*, 21(5):417–441, 1984.
- Michael R. Hansen and Zhou Chaochen. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9:283 – 330, 1997.

- Matthew Hennessy. Timed Process Algebras: A Tutorial. Lecture notes, International Summer School on Process Design Calculi, Martoberdorf, 1992.
- Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985. ISSN 0004-5411. doi: <http://doi.acm.org/10.1145/2455.2460>.
- Matthew Hennessy and Tim Regan. A Process Algebra for Timed Systems. *Information and Computation*, 117(2):221–239, 1995.
- Thomas A. Henzinger. It’s about time: Real-time logics reviewed. In *CONCUR’98 Concurrency Theory*, LNCS 1466, pages 439 – 454. Springer, 1998. doi: 10.1007/BFb0055640.
- Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. ISBN 3-540-37215-6.
- Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Control*, 111(2):193 – 244, June 1992.
- Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.
- Anders Hessel and Paul Pettersson. Model-Based Testing of a WAP Gateway: an Industrial Study. In *In Proceedings of FMICS and PDMC*, LNCS 4346, 2006.
- Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/360248.360251>.
- Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Press, 1997.
- Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.
- Hee-Hwan Kwak, Insup Lee, Anna Philippou, Jin-Young Choi, and Oleg Sokolsky. Symbolic schedulability analysis of real-time systems. *IEEE RTSS*, 1998.
- Leslie Lamport. Real Time is Really Simple. Technical Report MSR-TR-2005-30, Microsoft Research, 2005.

- Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal of Software Tools for Technology Transfer*, 1(1/2):134–152, December 1997.
- Viktor Leijon and Johan Nordlander. Classical scheduling and timed processes - the  $\text{Ti}\pi$  calculus. In *12th International Conference, FOSSACS Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, 2009*. **submitted**.
- Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice Hall, 2nd edition, 1998.
- Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.
- Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1989.
- Robin Milner. *Communicating and mobile systems: the  $\pi$ -calculus*. Cambridge University Press, 1999.
- Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *INFCTRL: Information and Computation (formerly Information and Control)*, 100, 1992a.
- Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *INFCTRL: Information and Computation (formerly Information and Control)*, 100, 1992b.
- Faron Moller and Chris Tofte. A temporal calculus of communicating systems. In *Proceedings of CONCUR'90, LNCS 458*, pages 401–415, 1990.
- Uwe Nestmann. What is a "Good" Encoding of Guarded Choice? *Information and Computation*, 156(1-2):287–319, 2000.
- Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119, pages 179–194, Pisa, Italy, 1996. Springer-Verlag.
- Xavier Nicollin and Joseph Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.

- Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In *Proc. of CAV'91*, July 1991.
- John O'Donnell, Cordelia Hall, and Rex Page. *Discrete Mathematics Using a Computer*, chapter Equational Reasoning. Springer London, 2nd edition, 2006. ISBN 978-1-84628-241-6. doi: 10.1007/1-84628-598-4\_2.
- Joël Ouaknine and Steve Schneider. Timed CSP: A Retrospective. *Electronic Notes in Theoretical Computer Science*, 162:273–276, 2006.
- José Carlos. Palencia and Michael González Harbour. Offset-based response time analysis of distributed systems scheduled under edf. In *Proceedings of 15th Euro-micro Conference on Real-Time Systems (ECRTS'03)*. IEEE, 2003.
- Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. Available electronically, 1997.
- George M. Reed. A hierarchy of domains for real-time distributed computing. *Proceedings of the fifth international conference on Mathematical foundations of programming semantics table of contents*, pages 80–125, 1990.
- George M. Reed and Andrew W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988. ISSN 0304-3975. doi: [http://dx.doi.org/10.1016/0304-3975\(88\)90030-8](http://dx.doi.org/10.1016/0304-3975(88)90030-8).
- Davide Sangiorgi. On the origins of bisimulation, coinduction, and fixed points. Technical Report 2007-24, Department of Computer Science, University of Bologna, 2007.
- Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.
- Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, John Caccamo, Marcoand Lehoczky, and Aloysius K Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2):101–155, 2004.
- Joseph Sifakis. Modeling real-time systems-challenges and work directions. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 373–389. Springer, 2001. ISBN 3-540-42673-6.
- John A. Stankovic, Insup Lee, Aloysius Mok, and Raj Rajkumar. Opportunities and obligations for physical computing systems. *IEEE Computer*, 38(11):23–31, 2005.

UPPAAL. Tool Environment for Validation and Verification of Real-Time Systems, July 2007. Fetched from <http://www.it.uu.se/research/group/darts/papers/texts/uppaal-pamphlet.pdf>.

Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Chalmers University of Technology, Dept. of Computer Sciences, 1991.