Electronic Notes in Theoretical Computer Science 52 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume52.html pages

Towards a Process Algebra for Shared Processors

Mikael Buchholtz, Jacob Andersen, and Hans Henrik Løvengreen

Informatics and Mathematical Modelling
Technical University of Denmark
DK–2800 Lyngby, Denmark

Abstract

We present initial work on a timed process algebra that models sharing of processor resources allowing preemption at arbitrary points in time. This enables us to model both the functional and the timely behaviour of concurrent processes executed on a single processor. We give a refinement relation that describes that one process is more deterministic than another. Applications of the model for process scheduling, programming language semantics, and kernel development are outlined.

1 Introduction

To argue about correctness of a real-time system both the functional and the timely behaviour of the system have to be taken into account. One abstraction of a system is to view it as a number of concurrent processes. In many real systems this abstraction will be realised by executing the processes on a shared processor where execution is switched among the processes. For such a system, the fact that processes have to share the processor cannot be ignored, since it may influence both the timely and the functional behaviour of the system.

Systems of processes sharing a common processor is well described in the theory of scheduling, see e.g. [13]. However, scheduling theory generally abstracts from the actual behaviour of processes and hence cannot be used to analyse functional properties of a system.

On the other hand, operational models such as automata and processes algebras may describe the functional behaviour of a system in detail. Today, extension of these models with a notion of time is well-established, but is usually based upon an assumption of indefinite computational resources being available.

Recently, some authors have studied process scheduling in the context of timed automata [8,1]. These approaches, however, seem focused on modelling scheduling and they have no direct way to combine their model of the process scheduling with a system model where the behaviour of processes is described in detail.

An alternative is to take a process algebraic approach in which systems, from a modelling point of view, can be specified by writing programs in a programming language. However, timed process algebras typically assume maximal parallelism, which may be interpreted as if each process had its own dedicated processor. Therefore, they cannot adequately model real-time systems where processes share a processor. Under the assumption of maximal parallelism, you may say that the processor will always be available to a process. This means that the passing of time may be described without considering whether the processor is available or not. In many process algebras this leads to modelling time as transitions like $P \xrightarrow{t} P'$, stating that the process P changes into P' while t units of time pass.

To remedy the problem that this model of time cannot be used as a model of systems with a shared processor, we propose a new model. Our simple idea is to say that a process may let time pass in two different ways, which gives rise to transitions of the following two types

- $P \xrightarrow{\delta(t)} P'$ stating that t units of time pass, *without* the process using the processor.
- $P \xrightarrow{\rho(t)} P'$ stating that t units of time pass, while the process *does use* the processor.

In this model we are able to express that one process preempts another process currently using the processor. Thus, we are able to describe scheduling of processes in accordance with standard scheduling theory.

In this paper we present work-in-progress on a process algebra which uses the above model as the underlying model of time. We focus on processes sharing a single processor scheduled according to some scheduling strategy. The algebra allows modelling of multi-level scheduling as found in modern operating systems. Multi-processor systems can be described but only with static allocation of processes to processors. Extending this setup to a more general setting is left for future work.

This paper is organised as follows: First, we present the process algebra, giving its syntax and semantics. Second, we describe how different scheduling policies may be implemented. Third, we describe a number of properties, which the process algebra fulfil and we give a refinement relation between processes. Fourth, we give some examples of applications in which the process algebra has been used, followed by brief comments on related works. As this paper presents work-in-progress, we conclude by outlining future work.

2 Syntax and Semantics

In this section we present the process algebra. First, we give the syntax and a short informal semantics. Second, we give a formal operational semantics of the algebra.

The process algebra defines processes from the syntactic category Proc with P ranging over it. In general, when we define a variable to range over some set we also let any primed or indexed versions of the variables range over the same set. The syntax of a process is given by the grammar

$$P ::= P_1 \parallel P_2 \;\mid\; P_1 + P_2 \;\mid\; \mathbf{0} \;\mid\; \mathit{rec}\ X : P \;\mid\; X \;\mid$$
$$e.P \;\mid\; P \setminus C \;\mid\; \mathit{id} :: P \;\mid\; \langle\!\langle P \rangle\!\rangle_S \;\mid\; [P]$$

Processes may be combined in parallel compositions denoted by $P_1 \parallel P_2$. $P_1 + P_2$ is non-deterministic choice between P_1 and P_2. $\mathbf{0}$ is the terminated process. Standard recursion is written $\mathit{rec}\ X : P$ where X is a process variable from some set PVar. We define the set Proc to be the subset of Proc where all process variables are bound i.e. only appear inside a recursive process, where they have been defined.

$e.P$ is prefix of an event e to the process P. Events belong to the set $E = \mathit{UTE} \cup \mathit{TE}$ of untimed events and timed events. Untimed events, with u ranging over them, take no time and may either be the internal action τ or synchronisation events $c \in \mathcal{C} \cup \overline{\mathcal{C}}$ belonging to a set of names and co-names. We have that $c = \overline{\overline{c}}$ for any $c \in \mathcal{C} \cup \overline{\mathcal{C}}$ and $c \in \mathcal{C}$ if and only if $\overline{c} \in \overline{\mathcal{C}}$. Two processes may synchronise by performing the matching events c and $\overline{c}$, respectively. $P \setminus C$ with $C \subseteq (\mathcal{C} \cup \overline{\mathcal{C}})$ is used ensure that synchronisation of all events in C is restricted to the process P.

We use the real numbers, $\mathbb{R}$, to model time and let t range over this set. Timed events belong to the set $\mathit{TE} \overset{\mathit{def}}{=} \{\delta(t_l, t_u), \rho(t_l, t_u) \mid t_l \in \mathbb{R} \wedge\ t_u \in \mathbb{R}_{>0} \wedge t_l \leq t_u\}$. The timed event $\delta(t_l, t_u)$ means that the process *delays* i.e. does not request processor time for a period between a lower bound of t_l time units and an upper bound of t_u time units. $\rho(t_l, t_u)$ means that the process *requests* between t_l and t_u units of processor time in order to continue.

In order to distinguish and schedule processor requests, we use a scheme of nested names. A name $p \in \mathit{Name}$ is a sequence of identifiers determining one or more sub-processes within a process. Initially, all processes implicitly have the anonymous, empty name ϵ associated with them. The naming construct $\mathit{id} :: P$ declares the identifier $\mathit{id} \in \mathit{ID}$ to be an explicit name for the process P. Thereby, any sub-processes of P will get id prefixed to their names. Names are typically used to designate concurrent sub-processes as in the process $a :: P_1 \parallel b :: \tau.(x :: P_2 \parallel y :: P_3)$. For this process, the name a refers to P_1 and the names $b.x$ and $b.y$ refer to P_2 and P_3, respectively.

Names are used when access to the processor is scheduled. This is done by the construct $\langle\!\langle P \rangle\!\rangle_S$, where $S \in \mathit{Sch}$ is a scheduler. Different scheduling policies may be enforced by changing the scheduler component S.

The final construct is the closing operator $[P]$ which confines the sharing of processor time to the process P. Thus, $[P]$ may be seen as assigning a dedicated processor to the process P.

We give the semantics for the processes by defining a labelled transition system $\langle Proc, L, \longrightarrow\rangle$. The labels are given as the set $L \stackrel{def}{=} UTL \cup TL$ of untimed labels and timed labels; l ranges over L. We take $UTL \stackrel{def}{=} UTE$. All transitions describing internal actions or completed synchronisations will be labelled by τ. We ensure that internal actions as well as all possible synchronisation between processes is urgent, i.e. takes place before time passes. We refer to this property as τ-*urgency*. Uncompleted synchronisation is labelled by the name of the synchronisation event. The set TL contains timed labels and is defined as $TL \stackrel{def}{=} \{\delta(t), \rho_p(t) \mid t \in \mathbb{R}_{>0} \wedge p \in Name\}$. Transitions labelled $\delta(t)$ denote that t time units passes without using the processor while a label $\rho_p(t)$ denote that the (sub-)process named p uses the processor for t time units. Note that these timed labels always describe that a positive amount of time passes.

The transition relation is defined as the smallest relation satisfying the inference rules given in the following. As it is standard, premises and the conclusion of the rules matches all occurrences of transitions on the form $P \xrightarrow{l} P'$, thus, making quantisation of the variable P, P', and l implicit. In addition, we use the following shorthand notation:

$$P \xrightarrow{l} \;\stackrel{def}{=}\; \exists P' \bullet P \xrightarrow{l} P' \qquad\qquad P \xrightarrow{\delta} \;\stackrel{def}{=}\; \exists t \bullet P \xrightarrow{\delta(t)}$$

$$P \stackrel{l}{\not\rightarrow} \;\stackrel{def}{=}\; \neg(P \xrightarrow{l}) \qquad\qquad P \xrightarrow{\rho_p} \;\stackrel{def}{=}\; \exists t \bullet P \xrightarrow{\rho_p(t)}$$

These conventions are used throughout this paper.

Prefix:

$$\tau.P \xrightarrow{\tau} P$$

$$c.P \xrightarrow{c} P \qquad\qquad c.P \xrightarrow{\delta(t)} c.P$$

$$\delta(t_l, t_u).P \xrightarrow{\delta(t)} P \qquad \textbf{if } t_l \leq t \leq t_u$$

$$\delta(t_l, t_u).P \xrightarrow{\delta(t)} \delta(t_l - t, t_u - t).P \qquad \textbf{if } t < t_u$$

$$\rho(t_l, t_u).P \xrightarrow{\rho_\epsilon(t)} P \qquad \textbf{if } t_l \leq t \leq t_u$$

$$\rho(t_l, t_u).P \xrightarrow{\rho_\epsilon(t)} \rho(t_l - t, t_u - t).P \qquad \textbf{if } t < t_u$$

$$\rho(t_l, t_u).P \xrightarrow{\delta(t)} \rho(t_l, t_u).P$$

τ events take no time. Synchronisation may either be performed instantaneously or postponed without using the processor. When processes are joined by parallel composition, as we will see shortly, the synchronisation is made urgent. Remember that $t > 0$ in labels $\delta(t)$. Thus, A delay event $\delta(t_l, t_u)$ is required to use time but will never use the processor. All the time may pass in one transition, or it may be divided into several transitions. The request for processor time behaves in the same way, but uses the processor when time passes. Furthermore, if the process is preempted it will let time pass, but its request for processor time remains.

Choice:

$$\frac{P_1 \xrightarrow{u} P_1'}{P_1 + P_2 \xrightarrow{u} P_1'} \qquad \frac{P_2 \xrightarrow{u} P_2'}{P_1 + P_2 \xrightarrow{u} P_2'}$$

$$\frac{P_1 \xrightarrow{\rho_p(t)} P_1' \quad P_2 \xrightarrow{\delta}}{P_1 + P_2 \xrightarrow{\rho_p(t)} P_1'} \qquad \frac{P_1 \xrightarrow{\delta} \quad P_2 \xrightarrow{\rho_p(t)} P_2'}{P_1 + P_2 \xrightarrow{\rho_p(t)} P_2'}$$

$$\frac{P_1 \xrightarrow{\delta(t)} P_1' \quad P_2 \xrightarrow{\delta(t)} P_2'}{P_1 + P_2 \xrightarrow{\delta(t)} P_1' + P_2'}$$

If either one of the processes in the choice can perform an untimed event, u, the whole construct proceeds as that process. If one of the processes can use processor time this branch is chosen but only if the other process can let time pass. Thus, we make sure that if the second process can perform a τ-transition then the first process cannot let time pass. Finally, if both processes are be able to delay, the choice will be postponed.

Recursion:

$$\frac{P[X \leftarrow\!\!\dashv \textit{rec}\ X : P] \xrightarrow{l} P'}{\textit{rec}\ X : P \xrightarrow{l} P'}$$

Recursion is done by standard unfolding, where $P[X \leftarrow\!\!\dashv Q]$ denotes the process P with any free occurrences of the process variable X substituted with the process Q.

Stop:

$$\mathbf{0} \xrightarrow{\delta(t)} \mathbf{0}$$

Even a terminated process cannot prevent time from passing.

Restriction:

$$\frac{P \xrightarrow{l} P'}{P \setminus C \xrightarrow{l} P' \setminus C} \quad \textbf{if } l \in (\mathcal{C} \cup \overline{\mathcal{C}}) \Rightarrow (l \notin C \wedge \bar{l} \notin C)$$

Naming:

$$\frac{P \xrightarrow{\rho_p(t)} P'}{id :: P \xrightarrow{\rho_{id.p}(t)} id :: P'}$$

$$\frac{P \xrightarrow{l} P'}{id :: P \xrightarrow{l} id :: P'} \quad \textbf{if } l \in L \backslash \{\rho_p(t) \mid t \in \mathbb{R}_{>0} \wedge p \in \mathit{Name}\}$$

Names are only used for scheduling of processor time, so only ρ-transitions contain information of process names. In the naming construct, id is prefixed to the name of the process P.

Parallel composition:

$$\frac{P_1 \xrightarrow{u} P_1'}{P_1 \parallel P_2 \xrightarrow{u} P_1' \parallel P_2} \qquad \frac{P_2 \xrightarrow{u} P_2'}{P_1 \parallel P_2 \xrightarrow{u} P_1 \parallel P_2'}$$

$$\frac{P_1 \xrightarrow{c} P_1' \quad P_2 \xrightarrow{\overline{c}} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'}$$

$$\frac{P_1 \xrightarrow{\rho_p(t)} P_1' \quad P_2 \xrightarrow{\delta(t)} P_2' \quad P_1 \parallel P_2 \stackrel{\tau}{\not\rightarrow}}{P_1 \parallel P_2 \xrightarrow{\rho_p(t)} P_1' \parallel P_2'} \qquad \frac{P_1 \xrightarrow{\delta(t)} P_1' \quad P_2 \xrightarrow{\rho_p(t)} P_2' \quad P_1 \parallel P_2 \stackrel{\tau}{\not\rightarrow}}{P_1 \parallel P_2 \xrightarrow{\rho_p(t)} P_1' \parallel P_2'}$$

$$\frac{P_1 \xrightarrow{\delta(t)} P_1' \quad P_2 \xrightarrow{\delta(t)} P_2' \quad P_1 \parallel P_2 \stackrel{\tau}{\not\rightarrow}}{P_1 \parallel P_2 \xrightarrow{\delta(t)} P_1' \parallel P_2'}$$

An untimed events may be performed in the parallel composition if either of the constituents can perform it. For time to pass in the parallel composition it is required that both of its constituents are able to let it do so. Furthermore, the rules ensure τ-urgency. The rules for parallel composition also ensure that at most one process is allowed to use the processor at a time.

Closure:

$$\frac{T \xrightarrow{u} T'}{[T] \xrightarrow{u} [T']}$$

$$\frac{T \xrightarrow{\rho_p(t)} T'}{[T] \xrightarrow{\delta(t)} [T']} \qquad \frac{T \xrightarrow{\delta(t)} T' \quad \forall p \bullet T \stackrel{\rho_p}{\not\rightarrow}}{[T] \xrightarrow{\delta(t)} [T']}$$

When a process is closed under processor time consumption no further processor time is needed. Additionally, the process may only delay if it cannot use processor time.

Scheduling:

The scheduling operator is elaborated in the next section.

3 Scheduling

We would like our approach to scheduling to reflect two main characteristics of scheduling in modern operating systems: (i) That the scheduling strategy can be defined in a way independent from the rest of the kernel, and (ii) that scheduling is often done at several levels of the system.

We have tried to achieve these goals by making scheduling a general operator $\langle\!\langle\cdot\rangle\!\rangle_S$ indexed by a scheduler implementing a particular strategy. This way, the scheduler becomes a plug-in component and different schedulers may be used at different levels.

A scheduler normally schedules a set of processes composed in parallel. In order to identify the various processes, the scheduler uses *process names*. Usually each process is assumed to be given a unique name, but this needs not be the case. If several processes share a name (e.g. the empty name ϵ), they become a process group, which is internally unscheduled. Also, when a process forks into sub-processes, these may be given sub-names enabling the scheduler to treat them in a special way, e.g. as threads within a process.

The scheduler is given by a transition system that must satisfy a few behavioural properties. The transition rules for the scheduling operator, conveys the interaction between the scheduler and the scheduled process. Mainly, the scheduler determines which process should be allowed to use the processor by engaging in events of the form $\sigma_p(t)$ indicating that the process named p uses the processor for t time units. However, in order to know which processes to schedule, the scheduler must be informed about processes becoming ready to execute as well as processes blocking execution. This is here accomplished through special *scheduling events* κ_p^+ and κ_p^- that indicate that the process named p becomes ready to run (i.e. use processor time) or blocks (i.e. cannot use processor time), respectively. Furthermore, the scheduler may perform internal events denoted by τ and use processor time itself denoted by $\rho(t)$. Finally, the scheduler may idle denoted by $\delta(t)$-events.

3.1 Scheduler

Definition 3.1 A *scheduler* S is an (initialised) labelled transition system $\langle \Sigma, L_{Sch}, \longrightarrow, s_0\rangle$ where

Σ	Is a set of scheduler states
L_{Sch}	$= \{\tau\} \cup \{\delta(t), \rho(t), \sigma_p(t), \kappa_p^+, \kappa_p^- \mid p \in \mathit{Name} \wedge t \in \mathbb{R}_{>0}\}$
$\longrightarrow$	$\subseteq \Sigma \times L_{Sch} \times \Sigma$
s_0	$\in \Sigma$ is the initial state.

The transition relation has to fulfil two *scheduler requirements* for all $s, s' \in \Sigma$

sr1) $s \xrightarrow{\rho} \vee\, s \xrightarrow{\sigma_p} \;\Rightarrow s \xrightarrow{\delta}$

sr2) $s \xrightarrow{\delta(t)} s' \;\Rightarrow \forall t' \in]0, t[\ \exists s'' \bullet s \xrightarrow{\delta(t')} s'' \wedge s'' \xrightarrow{\delta(t-t')} s'$

$s \xrightarrow{\rho(t)} s' \;\Rightarrow \forall t' \in]0, t[\ \exists s'' \bullet s \xrightarrow{\rho(t')} s'' \wedge s'' \xrightarrow{\rho(t-t')} s'$

$s \xrightarrow{\sigma_p(t)} s' \Rightarrow \forall t' \in]0, t[\ \exists s'' \bullet s \xrightarrow{\sigma_p(t')} s'' \wedge s'' \xrightarrow{\sigma_p(t-t')} s'$

If $S = \langle \Sigma, L_{Sch}, \longrightarrow, s\rangle$ and $S' = \langle \Sigma, L_{Sch}, \longrightarrow, s'\rangle$ we write $S \xrightarrow{l} S'$ iff $s \xrightarrow{l} s'$. The set of schedulers is denoted Sch.

The scheduler requirement sr1) ensures that the scheduler itself is preemtable. Requirement sr2) ensures interpolation of any timed event, which is needed for compositionality.

Given the notion of a scheduler, we can now define the transition relation for the scheduling operator:

$$\frac{P \xrightarrow{u} P'}{\langle\!\langle P\rangle\!\rangle_S \xrightarrow{u} \langle\!\langle P'\rangle\!\rangle_S} \qquad \frac{S \xrightarrow{\tau} S'}{\langle\!\langle P\rangle\!\rangle_S \xrightarrow{\tau} \langle\!\langle P\rangle\!\rangle_{S'}}$$

$$\frac{P \xrightarrow{\rho_p} \quad S \xrightarrow{\kappa_p^+} S'}{\langle\!\langle P\rangle\!\rangle_S \xrightarrow{\tau} \langle\!\langle P\rangle\!\rangle_{S'}} \qquad \frac{P \overset{\rho_p}{\not\rightarrow} \quad P \xrightarrow{\delta} \quad S \xrightarrow{\kappa_p^-} S'}{\langle\!\langle P\rangle\!\rangle_S \xrightarrow{\tau} \langle\!\langle P\rangle\!\rangle_{S'}}$$

$$\frac{P \xrightarrow{\rho_p(t)} P' \quad S \xrightarrow{\sigma_p(t)} S' \quad \langle\!\langle P\rangle\!\rangle_S \overset{\tau}{\not\rightarrow}}{\langle\!\langle P\rangle\!\rangle_S \xrightarrow{\rho_\epsilon(t)} \langle\!\langle P'\rangle\!\rangle_{S'}} \qquad \frac{P \xrightarrow{\delta(t)} P' \quad S \xrightarrow{\rho(t)} S' \quad \langle\!\langle P\rangle\!\rangle_S \overset{\tau}{\not\rightarrow}}{\langle\!\langle P\rangle\!\rangle_S \xrightarrow{\rho_\epsilon(t)} \langle\!\langle P'\rangle\!\rangle_{S'}}$$

$$\frac{P \xrightarrow{\delta(t)} P' \quad S \xrightarrow{\delta(t)} S' \quad \langle\!\langle P\rangle\!\rangle_S \overset{\tau}{\not\rightarrow}}{\langle\!\langle P\rangle\!\rangle_S \xrightarrow{\delta(t)} \langle\!\langle P'\rangle\!\rangle_{S'}}$$

Note that any τ-event caused either by the process, by the scheduler, or by scheduling events is made urgent. Note also that the scheduling operator hides the process names of the scheduled processes.

3.2 Fixed Priority Scheduling

Fixed priorities is a widely used scheduling strategy for which a well-developed body of scheduling theory is available. We assume that we are given a mapping of process names to numbers:

$$prio : Name \rightarrow \mathbb{N}$$

Now, a fixed priority scheduler is given by $S_{\mathrm{FP}} \overset{def}{=} \langle \mathcal{P}(Name), L_{Sch}, \longrightarrow, \emptyset\rangle$ where a scheduler state R is a set of names of processes that are ready and where the transition relation is given by:

$$R \xrightarrow{\delta(t)} R$$

$$R \xrightarrow{\sigma_p(t)} R' \qquad \textbf{if } p \in R \wedge \forall\, p' \in R \bullet \mathit{prio}(p') \leq \mathit{prio}(p)$$

$$R \xrightarrow{\kappa_p^+} R \cup \{p\} \qquad \textbf{if } p \notin R$$

$$R \xrightarrow{\kappa_p^-} R \backslash \{p\} \qquad \textbf{if } p \in R$$

The transition relation is readily seen to fulfil the two scheduler requirements. The σ-rule together with the κ^+-rule ensure that a process that becomes ready to run is scheduled immediately if it has the highest priority. Note that the κ^+-rule dynamically "recognises" nested sub-processes when they become ready to run if they are uniquely named. The side conditions of the κ-rules prevent infinite sequences of scheduling events to be performed.

3.3 Round-Robin Scheduling

Another classic scheduling strategy is round-robin in which the processes are executed cyclically giving each process a bounded time-slot of size T. A round-robin scheduler can be defined as $S_{\mathrm{RR}} \stackrel{\mathit{def}}{=} \langle \mathit{Name}^* \times \mathbb{R}_{\geq 0}, L_{Sch}, \longrightarrow, \langle [\,], 0 \rangle \rangle$ with states of the form $\langle Q, r \rangle$ where Q is a list of ready processes and r is the remainder of the current time-slot. We use $[\,]$ as the empty list and $\hat{}$ as concatenation of lists and elements and define the transition relation by:

$$\langle [\,], 0 \rangle \xrightarrow{\tau} \langle [\,], T \rangle$$

$$\langle p \,\hat{}\, Q', 0 \rangle \xrightarrow{\tau} \langle Q' \,\hat{}\, p, T \rangle$$

$$\langle Q, r \rangle \xrightarrow{\delta(t)} \langle Q, r - t \rangle \qquad \textbf{if } 0 < t \leq r$$

$$\langle p \,\hat{}\, Q', r \rangle \xrightarrow{\sigma_p(t)} \langle p \,\hat{}\, Q', r - t \rangle \qquad \textbf{if } 0 < t \leq r$$

$$\langle p \,\hat{}\, Q', r \rangle \xrightarrow{\kappa_p^-} \langle Q', r \rangle$$

$$\langle Q, r \rangle \xrightarrow{\kappa_p^+} \langle Q \,\hat{}\, p, r \rangle \qquad \textbf{if } p \notin Q$$

Again the transition relation fulfils the two scheduling requirements. Note that in case of at process blocking before its time-slot has ended (κ^--rule), the next process on the queue just gets the remaining part of the slot. Other strategies can be described by variations of the rules.

3.4 Example: A Transputer-like Architecture

The Occam language was developed together with the transputer computer architecture [6]. The transputer had a built-in scheduler with two levels of priorities. The transputer had autonomous external links that could support synchronous communication. A model of an Occam program with a number of processes mapped to various transputers can be expressed as:

$$[\langle\!\langle \mathsf{high} :: P_1 \parallel \mathsf{high} :: P_2 \parallel \ldots \parallel \mathsf{low} :: P_n \rangle\!\rangle_{S_{\mathrm{FP}}}] \parallel \ldots \parallel [\langle\!\langle \ldots \rangle\!\rangle_{S_{\mathrm{FP}}}]$$

It may be noted that our model uses a notion of arbitrary preemption and thus does not allow for modelling the refined preemption scheme used by the real transputer in which a process can be preempted only at a few scheduling points such as communications and backward jumps.

3.5 Example: Multi-level Scheduling

The result of applying the scheduling operator is a process, which is itself subject to scheduling. This may be used to model multi-level scheduling systems. We illustrate this by an example that resembles the scheduling in Real-Time Linux [3]. Here, a standard operating system (Linux) is run as a low priority process in a system scheduled by fixed priorities. The operating system uses its own scheduler to schedule the processor time it obtains.

Here, we consider the case of scheduling a number of hard real-time processes $P_1, P_2, \ldots, P_n$ together with a number of "user processes" $U_1, U_2, \ldots U_m$ scheduled by round-robin. This can be described by the system:

$$[\langle\!\langle \mathsf{p}_1 :: P_1 \parallel \ldots \parallel \mathsf{p}_n :: P_n \parallel \mathsf{os} :: \langle\!\langle \mathsf{u}_1 :: U_1 \parallel \ldots \parallel \mathsf{u}_m :: U_m \rangle\!\rangle_{S_{\mathrm{RR}}} \rangle\!\rangle_{S_{\mathrm{FP}}}]$$

Here, S_{FP} is using a priority such that $\mathit{prio}(\mathsf{p}_n) > \mathit{prio}(\mathsf{os})$ for all n, such that os gets the lowest priority.

4 A Refinement Notion

A main goal of our approach is to be able to describe and verify the implementation of a real-time language by means of translation into code executed by an operating system kernel. A characteristic of this development path is that the timing of the program operations becomes more and more determined. Not only in the sense that timing parameters become more precise as code for an actual machine is generated, but also in the sense that a particular scheduling strategy will enable schedulability to be analysed.

For these reasons, we look for a process refinement notion rather than an equivalence notion. It is well-known that there is a wide spectrum of refinement notions for processes, see e.g. [16]. In this section we try to adopt one of these, the notion of $\frac{2}{3}$bisimulation, and show that it is compositional and adequate for simple refinements. To pave the way for this, we start by stating a number of properties of our model.

4.1 Model Properties

The first main characteristic of processes of the algebra is that they enjoy the natural property of allowing time interpolation:

Definition 4.1 A process P is *time interpolant* if

$$\begin{array}{rl} & P \xrightarrow{\delta(t)} P' \quad \Rightarrow \forall t' \in]0,t[\ \exists P'' \bullet P \xrightarrow{\delta(t')} P'' \wedge P'' \xrightarrow{\delta(t-t')} P' \\ \text{and} & P \xrightarrow{\rho_p(t)} P' \Rightarrow \forall t' \in]0,t[\ \exists P'' \bullet P \xrightarrow{\rho_p(t')} P'' \wedge P'' \xrightarrow{\rho_p(t-t')} P' \end{array}$$

Lemma 4.2 *All processes in Proc are time interpolant.*

Proof. Straightforward using structural induction on the processes in *Proc*. The case of the scheduling operation uses the fact that the scheduler itself is required to be time interpolant by scheduling requirement sr2). □

Thus, any timed transition may be broken into smaller steps as required for proper parallel composition. We note that the inverse property of time interpolation, which is sometimes called time additivity, does not hold. This is a design choice in the semantics, motivated by the fact that many meaningful refinements of processes can readily be expressed without the use of a time additivity property. We also note that processes in *Proc* are not time-deterministic in the sense that a process will always develop deterministically as a result of (a sequence of) timed transitions. This is due to the nature of the events $\delta(t_l, t_u)$ and $\rho(t_l, t_u)$, which by definition describes non-deterministic behaviour as a result of letting time pass.

The next characteristic of processes is that any process needing processor time may be preempted. Thus, a process also has to let time pass without using the processor. This is formalised as

Definition 4.3 A process P is *preemptable* if

$$P \xrightarrow{\rho_p} \;\Rightarrow P \xrightarrow{\delta}$$

Lemma 4.4 *All processes in Proc are preemptable.*

Proof. By structural induction using Lemma 4.2. In particular, the case involving the scheduling construct $\langle\!\langle P \rangle\!\rangle_S$ relies on S fulfilling scheduler requirement sr1). □

Some effort in the semantics of the process algebra has gone into ensuring τ-urgency. This property can be formalised as

Definition 4.5 A process P is *τ-urgent* if

$$P \xrightarrow{\tau} \;\Rightarrow P \overset{\delta}{\not\rightarrow} \wedge P \overset{\rho_p}{\not\rightarrow}$$

We state without proof

Lemma 4.6 *All processes in Proc are τ-urgent.*

By combining Lemma 4.4 and Lemma 4.6 we get that a process in *Proc* cannot do both δ and τ:

$$\neg(P \xrightarrow{\tau} \wedge P \xrightarrow{\delta})$$

4.2 Refinement

Our refinement relation is tentatively based on $\frac{2}{3}$bisimulation presented e.g. in [12] (also known as *ready simulation* or *GSOS trace congruence* cf. [16]). In our definition of the refinement relation we use a *sort function*, which for untimed labels is the identity function. For $\delta(t)$ and $\rho_p(t)$ labels, the sort function gives the set of all events of the same 'sort', but with arbitrary timing periods.

$$\begin{aligned} \mathrm{sort} &: L \to \mathcal{P}(L) \\ \mathrm{sort}(u) &\overset{def}{=} \{u\} \\ \mathrm{sort}(\delta(t)) &\overset{def}{=} \{\delta(t') \mid t' \in \mathbb{R}_{>0}\} \\ \mathrm{sort}(\rho_p(t)) &\overset{def}{=} \{\rho_p(t') \mid t' \in \mathbb{R}_{>0}\} \end{aligned}$$

Definition 4.7 A binary relation $\mathcal{R} \subseteq \mathit{Proc} \times \mathit{Proc}$ is a *label abstracted $\frac{2}{3}$bisimulation* if whenever $(P, Q) \in \mathcal{R}$ then

(i) $\forall P', l \bullet P \xrightarrow{l} P' \Rightarrow \exists Q' \bullet Q \xrightarrow{l} Q' \wedge (P', Q') \in \mathcal{R}$

(ii) $\forall Q', l \bullet Q \xrightarrow{l} Q' \Rightarrow \exists P', l' \in \mathrm{sort}(l) \bullet P \xrightarrow{l'} P'$

Definition 4.8 A process P *refines* Q or $P \sqsupseteq Q$ if and only if there exists a (label abstracted) $\frac{2}{3}$bisimulation $\mathcal{R}$ so that $(P, Q) \in \mathcal{R}$.

We also say that process Q *is refined by* process P, written $Q \sqsubseteq P$ when P *refines* Q.

Lemma 4.9 *If $P \sqsupseteq Q$ the processes can do the same sorts of transitions:*

$$\begin{aligned} P \xrightarrow{u} &\Leftrightarrow Q \xrightarrow{u} \\ P \xrightarrow{\delta} &\Leftrightarrow Q \xrightarrow{\delta} \\ P \xrightarrow{\rho_p} &\Leftrightarrow Q \xrightarrow{\rho_p} \end{aligned}$$

Proof. Follows from Definitions 4.7 and 4.8 □

Theorem 4.10 *All process operators are monotone with respect to $\sqsupseteq$. That is, for all $P \sqsupseteq Q$, $P_1 \sqsupseteq Q_1$,$P_2 \sqsupseteq P_2$, and schedulers S, the following holds:*

$$\begin{aligned} e.P &\sqsupseteq e.Q \\ id :: P &\sqsupseteq id :: Q \\ P_1 + P_2 &\sqsupseteq Q_1 + Q_2 \\ P_1 \parallel P_2 &\sqsupseteq Q_1 \parallel Q_2 \\ P \setminus C &\sqsupseteq Q \setminus C \\ \langle\!\langle P \rangle\!\rangle_S &\sqsupseteq \langle\!\langle Q \rangle\!\rangle_S \\ [P] &\sqsupseteq [Q] \end{aligned}$$

Proof (sketch) The proof for each operator is done by defining a candidate for a $\frac{2}{3}$bisimulation and then checking conditions (i) and (ii) of Definition 4.7 using transition induction. Here, we only show central parts of the proofs for parallel composition and for the scheduling operator.

Parallel composition Let $\mathcal{R}_i$ be the $\frac{2}{3}$bisimulation justifying $P_i \sqsupseteq Q_i$ for $i \in \{1,2\}$. Let $\widehat{\mathcal{R}} = \{(P_1 \parallel P_2, Q_2 \parallel Q_2) \mid (P_1, Q_1) \in \mathcal{R}_1 \wedge (P_2, Q_2) \in \mathcal{R}_2\} \cup \mathcal{R}_1 \cup \mathcal{R}_2$. We obviously have that $(P_1 \parallel P_2, Q_1 \parallel Q_2) \in \widehat{\mathcal{R}}$. We now check that $\widehat{\mathcal{R}}$ is a $\frac{2}{3}$bisimulation.

Condition (i): Assume that $P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2$ due to $P_1 \xrightarrow{\tau} P_1$. Since $P_1 \sqsupseteq Q_1$, we know that $Q_1 \xrightarrow{\tau} Q_1'$ such that $(P_1', Q_1') \in \mathcal{R}_1$. Therefore $Q_1 \parallel Q_2 \xrightarrow{\tau} Q_1' \parallel Q_2$ where $(P_1' \parallel P_2, Q_1' \parallel Q_2) \in \widehat{\mathcal{R}}$. Similar arguments apply for all other transitions of $P_1 \parallel P_2$.

Condition (ii): Assume that $Q_1 \parallel Q_2 \xrightarrow{\delta}$. This can only be the case if $Q_1 \xrightarrow{\delta}$ and $Q_2 \xrightarrow{\delta}$. From $P_i \sqsupseteq Q_i$ it follows by applying condition (ii) that $P_1 \xrightarrow{\delta}$ and $P_2 \xrightarrow{\delta}$. Since all processes enjoy τ-urgency we then also have $P_1 \overset{\tau}{\not\to}$ and $P_2 \overset{\tau}{\not\to}$. Applying the δ-rule for parallel composition, we conclude $P_1 \parallel P_2 \xrightarrow{\delta}$. Similar arguments apply for all other transitions of $Q_1 \parallel Q_2$.

Scheduling Let $\mathcal{R}$ be the $\frac{2}{3}$bisimulation justifying $P \sqsupseteq Q$. Let $\widehat{\mathcal{R}} = \{(\langle\!\langle P \rangle\!\rangle_S, \langle\!\langle Q \rangle\!\rangle_S) \mid S \in \mathit{Sch} \wedge (P, Q) \in \mathcal{R}\} \cup \mathcal{R}$. We obviously have that $(\langle\!\langle P \rangle\!\rangle_S, \langle\!\langle Q \rangle\!\rangle_S) \in \widehat{\mathcal{R}}$. We check that $\widehat{\mathcal{R}}$ is a $\frac{2}{3}$bisimulation:

Condition (i): Assume that $\langle\!\langle P \rangle\!\rangle_S \xrightarrow{\delta(t)} \langle\!\langle P' \rangle\!\rangle_{S'}$. Then we know that $P \xrightarrow{\delta(t)} P'$, $S \xrightarrow{\delta(t)} S'$, and $\langle\!\langle P \rangle\!\rangle_S \overset{\tau}{\not\to}$. From the τ-rules we then further have $S \overset{\tau}{\not\to}$. By $P \sqsupseteq Q$ we get that $Q \xrightarrow{\delta(t)} Q'$ such that $(P', Q') \in \mathcal{R}$. Thus, $Q \overset{\tau}{\not\to}$ by τ-urgency. Now assume $\langle\!\langle Q \rangle\!\rangle_S \xrightarrow{\tau}$. Since we know $Q \overset{\tau}{\not\to}$ and $S \overset{\tau}{\not\to}$, this could then only be the case if (a) $Q \xrightarrow{\rho_p} \wedge S \xrightarrow{\kappa_p^+}$ or (b) $Q \overset{\rho_p}{\not\to} \wedge S \xrightarrow{\kappa_p^-}$. However, due to $P \sqsupseteq Q$ condition (ii), in case (a) this would imply $P \xrightarrow{\rho_p}$ and together with $S \xrightarrow{\kappa_p^+}$ we would be able to conclude $\langle\!\langle P \rangle\!\rangle_S \xrightarrow{\tau}$ contradicting the above. Similarly for case (b). Thus, $\langle\!\langle Q \rangle\!\rangle_S \overset{\tau}{\not\to}$. Now we can finally use $Q \xrightarrow{\delta(t)} Q'$ and $S \xrightarrow{\delta(t)} S'$ to conclude $\langle\!\langle Q \rangle\!\rangle_S \xrightarrow{\delta(t)} \langle\!\langle Q' \rangle\!\rangle_{S'}$ where $(\langle\!\langle P' \rangle\!\rangle_{S'}, \langle\!\langle Q' \rangle\!\rangle_{S'})$. Similar extensive case analyses apply to all the other transitions of $\langle\!\langle P \rangle\!\rangle_S$.

Condition (ii): Suppose that $\langle\!\langle Q \rangle\!\rangle_S \xrightarrow{\tau}$ due to $S \xrightarrow{\kappa_p^-}$, $Q \overset{\rho_p}{\not\to}$, and $Q \xrightarrow{\delta}$. Then by $P \sqsupseteq Q$ and Lemma 4.9 we know that $P \overset{\rho_p}{\not\to}$ and $P \xrightarrow{\delta}$. Together with $S \xrightarrow{\kappa_p^-}$ we can conclude $\langle\!\langle P \rangle\!\rangle_S \xrightarrow{\tau}$. Again similar arguments can be given for all other transitions of $\langle\!\langle Q \rangle\!\rangle_S$. □

Thus, the refinement notion chosen allows for compositional refinement of processes. The following simple example indicates that the refinement is weak enough to allow for some desired refinements.

Example 4.11 We would like to show that a process $Q_0 = \rho(1,5).R$ is refined by a process with a more constrained initial processor requirement $P_0 = \rho(2,3).R$ (for any $R \in \mathit{Proc}$). This is accomplished by the relation

$$\mathcal{R} \stackrel{\mathit{def}}{=} \{(P_t, Q_t) \mid 0 \le t < 3\} \cup \{(T,T) \mid T \in \mathit{Proc}\}$$

where $P_t \stackrel{\mathit{def}}{=} \rho(2-t, 3-t).R$ and $Q_t \stackrel{\mathit{def}}{=} \rho(1-t, 5-t).R$. We see that P_t is the derivative of P_0 after having used t units of processor time and not yet committed to R (likewise for Q_t). We obviously have that (P_0, Q_0) belongs to $\mathcal{R}$. Further, the relation can be shown to be a $\frac{2}{3}$bisimulation. For instance, take $(P_t, Q_t) \in \mathcal{R}$ and assume $P_t \xrightarrow{\rho(t')} P'$. Then P' must be either R or $P_{t+t'}$. In the first case, $2 \le t + t' \le 3$ and therefore also $1 \le t + t' \le 5$. Thus we also have $Q_t \xrightarrow{\rho(t')} R$. In the second case, we know that $t + t' < 3$ and thus also $t + t' < 5$ such that $Q_t \xrightarrow{\rho(t')} Q_{t+t'}$. Similarly for the other cases.

This and other simple examples such as $\rho(1,3).\rho(2,3).R \sqsupseteq \rho(2,8).R$ indicate that we are on the right track. There are, however, other natural refinements that cannot be shown. For instance $\rho(2,3).R$ should refine $\rho(1,2).\rho(1,2).R$ but this is not the case using the refinement of Definition 4.8. Thus, a weaker relation is still needed and will be sought in our future work.

5 Applications

The process algebra has been developed while working on formal development of a real-time kernel for a shared processor [2]. In this section we give examples of applications for which our time model and the process algebra have been used during this work.

5.1 Semantics of a Programming Language

We have used the process algebra to give semantics to a small programming language, which resembles Hoare's CSP [10] or Occam [6]. Programs of the language are from the syntactic category *RProg* with R ranging over it and consist of a fixed number of sequential processes in parallel. Each process contains one statement Q from the syntactic category *Stm*. Programs are built from the grammar

$$R ::= R_1 \parallel R_2 \mid Q$$

$$Q ::= \mathtt{delay}\, t \mid c? \mid c! \mid Q_1 \mathtt{;} Q_2 \mid \begin{array}{l}\mathtt{[}\ \ c_1? \longrightarrow Q_1 \\ \mathtt{[]}\ c_2? \longrightarrow Q_2\ \mathtt{]}\end{array} \mid \begin{array}{l}\mathtt{[}\ \ c? \longrightarrow Q_1 \\ \mathtt{[]}\ \mathtt{timeout}\, t \longrightarrow Q_2\ \mathtt{]}\end{array}$$

Statements are (possibly a sequential composition of) delay, synchronous communication, guarded alternative, or communication with time-out.

Communications takes place on channels that in order to simplify the translation are names from $\mathcal{C}$. The function $\mathit{Chan} : \mathit{RProg} \to \mathcal{P}(\mathcal{C})$ returns the set of channels that the program may communicate on and can be defined on the structure of a program.

The language is a "real" programming language, so it does not explicitly describe how processor time is required to execute different language constructs. The semantics of the language is given as a translation from programs of the language into processes in the process algebra. This translation is done on the structure of the programs, translating each language construct separately.

To translate an entire program we use the function $\mathcal{T} : \mathit{RProg} \to \mathit{Proc}$ while the function $\mathcal{TR} : \mathit{RProg} \to \mathit{Proc}$ is used to translate parallel composition. The first function makes communication urgent by restricting synchronisation on the channels, which the program may use for communication. The second function is a straight forward mapping between corresponding constructs. More interesting is the translation of statements where we use the continuation style function $\mathcal{TS} : \mathit{Stm} \to \mathit{Proc} \to \mathit{Proc}$. For the translation of statements below the terminating process $\mathbf{0}$ is passed as continuation of an entire statement to describe that the statement terminates at the end.

$$\begin{aligned}
\mathcal{T}[\![R]\!] &\stackrel{def}{=} (\mathcal{TR}[\![R]\!]) \setminus \mathit{Chan}[\![R]\!] \\
\mathcal{TR}[\![R_1 \parallel R_2]\!] &\stackrel{def}{=} \mathcal{TR}[\![R_1]\!] \parallel \mathcal{TR}[\![R_2]\!] \\
\mathcal{TR}[\![Q]\!] &\stackrel{def}{=} \mathcal{TS}[\![Q]\!]\mathbf{0}
\end{aligned}$$

Now, a program R scheduled by a specific scheduler S' will the be described by the process $[\langle\!\langle \mathcal{T}[\![R]\!] \rangle\!\rangle_{S'}]$.

As the first example of the translation of statements, regard the construct $\mathtt{delay}\,t$, which delays for at least t time units. In the definition of the translation function the argument P is the process, which describes what happens after the execution of $\mathtt{delay}\,t$.

$$\mathcal{TS}[\![\mathtt{delay}\,t]\!]P \stackrel{def}{=} \rho(T_{delay}, T_{delay}).\delta(t - T_{delay}, t - T_{delay}).P$$

The translation gives that the construct first T_{delay} units of processor time to execute the statement and figure out for how long it has to delay. Afterwards the statement delays for the remaining time units.

Communication in the language is a synchronisation just as it is in the process algebra. Thus, the translation becomes quite simple.

$$\begin{aligned}
\mathcal{TS}[\![c\mathtt{?}]\!]P &\stackrel{def}{=} \rho(T_{lrc}, T_{urc}).c.P \\
\mathcal{TS}[\![c\mathtt{!}]\!]P &\stackrel{def}{=} \rho(T_{lrc}, T_{urc}).\overline{c}.P
\end{aligned}$$

Here, we imagine that we only have a lower bound, T_{lrc}, and an upper bound, T_{urc}, of the time it takes to get ready to perform communication. Second, communication itself is performed by synchronising on events in the process algebra. Sequential composition is done in the standard way for continuation style semantics

$$\mathcal{TS}[\![Q_1\,;Q_2]\!]P \stackrel{def}{=} \mathcal{TS}[\![S_1]\!]\mathcal{TS}[\![S_2]\!]P$$

The (binary) alternative statement using communication guards should proceed as the statement following the guard if communication is possible.

$$\mathcal{TS}\left[\!\!\left[\begin{array}{l}\texttt{[}\ c_1? \longrightarrow Q_1\\ \texttt{[]}\ c_2? \longrightarrow Q_2\ \texttt{]}\end{array}\right]\!\!\right] P \stackrel{def}{=} \rho(2 \cdot T_{lrc}, 2 \cdot T_{urc}).\begin{pmatrix} c_1.\mathcal{TS}[\![Q_1]\!]P \\ +\, c_2.\mathcal{TS}[\![Q_2]\!]P\end{pmatrix}$$

First, the processor is used for between $2 \cdot T_{lrc}$ and $2 \cdot T_{urc}$ units of time. This processor time is used to decide between which channels the choice will be performed. Afterwards the actual choice can take place. The translation becomes relatively simple, since the choice operator of the process algebra has exactly the semantics we desire for the selection between communication guards.

As a final example, we look at communication with time-out. The statement proceeds as Q_1 if communication is possible within t time units. Otherwise, a time-out occurs and the statement proceeds as Q_2.

$$\mathcal{TS}\left[\!\!\left[\begin{array}{l}\texttt{[}\ c? \longrightarrow Q_1\\ \texttt{[]}\ \texttt{timeout}\, t \longrightarrow Q_2\ \texttt{]}\end{array}\right]\!\!\right] P \stackrel{def}{=} \rho(T_{lto}, T_{uto}).\begin{pmatrix} c?.\mathcal{TS}[\![Q_1]\!]P \\ +\, \delta(t,t).\tau.\mathcal{TS}[\![Q_2]\!]P\end{pmatrix}$$

Note that if communication on the channel c is not ready, the choice may be postponed for up to t time units before a τ-event can occur. Since τ-events are urgent, the choice will be made after at most t time units.

We see from the examples of the translation that it provides a clear overview of timings of the different statements. For example, use of processor time may easily be added as the timed event $\rho(t_l, t_u)$. In [2] we have used the technique described here for a larger language, which contains among other things recursion and alternative with multiple branches. The translations required for these new language construct follow the ideas described above closely and introduces no substantial novelties.

5.2 Development of a Real-Time Kernel

In [2] we have used the ideas for kernel development from the ProCoS project [15] to develop a real-time kernel for use in small embedded mono processor systems. In this, we have used the model of time presented in this paper to describe how processes share the processor.

As depicted in Figure 1, the development of the kernel comprises three development levels, each of which describes the embedded system at a certain level of abstraction. At the topmost level, the Programming Language Level, the system is regarded as a number of concurrent processes described by a program in a CSP-like language. At the middle level, the Machine Language Level, the system is still regarded as a number of concurrent processes. However, each process is given in an assembler-like language with instructions consisting of an op-code and arguments. Each processes is executed at its own virtual machine so management of e.g. a program counter, stack, and store can be described in detail. At the lowest level, the Kernel Level, the system is described as one virtual machine which executes all the processes, explicitly switching execution among them. In this description there is a clear distinction of which parts of the virtual machine that describe process behaviour and which parts that are not relate directly to a process i.e. the parts that must be handled by the kernel.

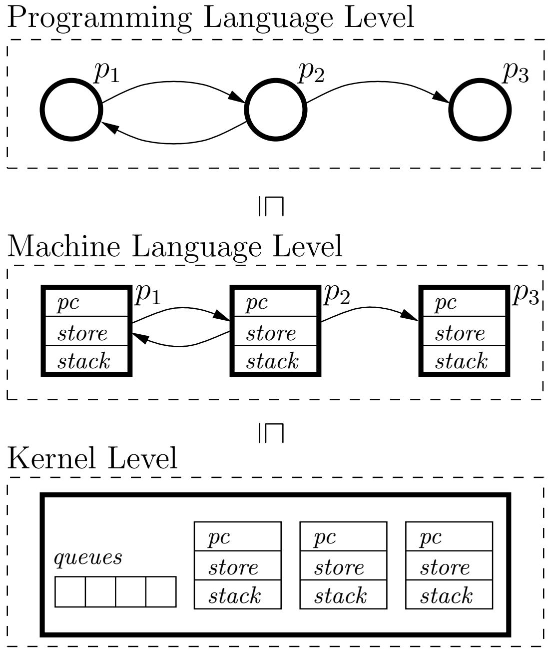

Fig. 1. Overview of kernel development levels for a system with three processes.

The behaviour of the system at each level is described by a labelled transition system. For the topmost level this transition system is given by an operational semantics of the CSP-like language according to the ideas described in section 5.1. At the other levels the transition systems are also defined by a number of inference rules. At all levels, we use the same model of time, thus having transition systems with actions and timed labels as described for the process algebra. In the development of the kernel we have dealt with at great number of aspects, which are outside the scope of this paper. These include: assignment of variables with evaluation of expressions, communication with value passing, an external interface using shared memory, kernel overhead, and timer interrupts.

Correctness of the system is based on a refinement relation, $\sqsupseteq$, between the transition systems describing two consecutive development levels. The refinement relation allows for a lower level to remove any non-determinism of an upper level. Except for this difference between their behaviours, the lower level *must* behave in the same way as the upper level.

6 Related Work

Related work is to be found in concurrency models that simultaneously address the issues of time, resource sharing, and priorities. Within process algebra, a number of approaches deal with time and priorities (see e.g. [7]). However, the only process algebraic approach that addresses all three issues seems to be [5]. In this algebra, the usage of a set A of resources for a period of t time units is denoted by a timed event A^t and its behaviour given by the rule

$$A^t.P \xrightarrow{A^{t'}} A^{t-t'}.P \quad \textbf{if } 0 < t' \leq t$$

similar to one of our $\rho(t)$ rules. However, they do not provide any means for allowing a partially completed resource usage to be temporarily preempted by another process.

In [4], process scheduling is studied within this framework, though a discrete time domain is used. Here, preemption of a process must be made explicitly so, whenever a process requests processor time a non-deterministic choice between processor use and idling is made and this is closed under processor sharing. Our approach is less general than the process algebra of [5], since we focus on processes executed under preemptive scheduling. On the other hand, we have been able to trade generality for simplicity when modelling parallel systems with processor sharing.

A novelty of our approach over the one of [5] very basically lies in the rule

$$\rho(t).P \xrightarrow{\delta(t')} \rho(t).P$$

allowing for the arbitrary preemption found in real-time operating systems. A similar rule is found in [9]. Here a fixed number of sequential processes may be composed in parallel and scheduled under some scheduling policy. Specifications of the schedulers is given as a function, which for a given set of processes will choose the processes that should run.

Our work has been inspired by the approach of [17] that distinguishes between *time passing* and (processor) *time consumption*. Their underlying behaviour model, however, does not have a notion of instantaneous events and they use logical characterisation of models rather than taking a constructive approach. The wealth of results and techniques developed for models based on transition systems, therefore, cannot immediately be related to this work.

An approach that is similar in spirit to ours, although in a totally different setting, is presented in [14]. There, transitions systems are represented in the logical formalism of TLA allowing for refinement by logical implication. Also scheduling may expressed compositionally by conjoining scheduling constraints. Each action is assumed to have a pre-execution phase where is requires a certain amount of processor time, before it can be executed. During the pre-execution phase, the action may be arbitrarily preempted as in our approach. They do not, however, consider multi-level scheduling.

7 Future Work

In section 3.1 we have described how a specific scheduling policies may be expressed by defining a new scheduler transition systems for each policy. However, the interface between the scheduler and the processes is fixed by the semantics of the scheduling construct and in its present form it is e.g. not possible to let processes dynamically change their priority. While this could easily be mended for any particular problem by altering the semantics of our scheduling construct, we need further studies on how to generalise interaction between processes and the scheduler.

Also, the model should be generalised to handle multi-processors and perhaps other preemptable resources. For the latter part, ideas from [5] could be applied.

A task lies in further studies of the theoretical implications of the process algebra. Some of this ground has already been covered and the results are satisfactory for "natural" processes arising from program semantics but further studies are needed e.g. on divergence, non-zenoness, time-stop, and recursion.

Traditional bisimulation equivalence has shown to be too strong for our kernel development application. Instead we have focused on refinement relations and have here presented one such relation, which is *compositional*, i.e. distribute over parallel composition (as well as over other operators). It has proven to be a non-trivial task to achieve compositionality due to the non-monotonic nature of resource sharing and urgency. We have successfully tried to adopt the relatively strong refinement relation: $\frac{2}{3}$bisimulation. Further work could be done on finding a weaker relation still fulfilling our requirements.

One of the goals of our work is to link resource-aware models like the one presented here, with standard timed models (e.g. timed automata) making them amenable for analysis by tools such as Uppaal [11] and others. One way to accomplish this would be to derive a timed automaton from a processor constrained transition system expressed in our model.

References

[1] Altisen, K., G. Gossler and J. Sifakis, *Scheduling - a methodology for the construction of scheduled systems*, Lecture Notes in Computer Science **1926** (2000), pp. 106–120.

[2] Andersen, J. and M. Buchholtz, "Formal Development of a Real-Time Kernel," Master's thesis, Department of Information Technology, Technical University of Denmark (2001).
URL `http://www.imm.dtu.dk/people/mib/masters/`

[3] Barabanov, M. and V. Yodaiken, *Real-time Linux*, Linux Journal (1996).

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

[5] Brémond-Grégoire, P. and I. Lee, *A process algebra of communicating shared resources with dense time and priorities*, Theoretical Computer Science (1997), pp. 179–219.

[6] Burns, A., "Programming in Occam-2," Addison-Wesley, 1988.

[7] Cleaveland, R., G. Löttgen and V. Natarajan, *Priority in process algebra*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, North-Holland, 2001 pp. 711–765.

[8] Fersman, E., P. Petterson and Wang Yi, *Timed automata with asynchronous processes: Schedulability and decidability*, in: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS2002)*, number 2280 in Lecture Notes of Computer Science, 2002, pp. 67–82.

[9] Fredette, A. N. and R. Cleaveland, *RTSL: A language for real-time scheduability analysis*, in: *Proceedings. Real-Time Systems Symposium*, 1993, pp. 274–283.

[10] Hoare, C. A. R., *Communicating sequential processes*, Communications of the ACM **21** (1978), pp. 666–677.

[11] Larsen, K. G., P. Pettersson and Wang Yi, Uppaal *in a Nutshell*, Int. Journal on Software Tools for Technology Transfer **1** (1997), pp. 134–152.

[12] Larsen, K. G. and A. Skou, *Bisimulation through probabilistic testing*, Information And Computation **94** (1991), pp. 1–29.

[13] Liu, J. W. S., "Real-Time Systems," Prentice Hall, 2000.

[14] Liu, Z. and M. Joseph, *Verification, refinement and scheduling of real-time programs*, Theoretical Computer Science **253** (2001), pp. 119–152.

[15] Søgaard-Andersen, J. F., C. Ø. Rump and H. H. Løvengreen, *A systematic kernel development*, SIGSOFT Software Engineering Notes **16** (1991), pp. 55–65.

[16] van Glabbeek, R. J., *The linear time – branching time spectrum I. The semantics of concrete, sequential processes*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, North-Holland, 2001 pp. 3–99.

[17] Zhou Chaochen, M. R. Hansen, A. P. Ravn and H. Rischel, *Duration specifications for shared processors*, in: *Formal Techniques in Real-Time and Fault-Tolerant Systems, Second International Symposium Proceedings*, 1991, pp. 21–32.