# Describing a Signal Analyzer in the Process Algebra PMC – A Case Study*

Henrik Reif Andersen Michael Mendler

**Abstract**

In this paper we take a look at real-time systems from an implementation-oriented perspective. We are interested in the formal description of genuinely distributed systems whose correct functional behaviour depends on real-time constraints. The question of how to combine real-time with distributed processing in a clean and satisfactory way is the object of our investigation.

The approach we wish to advance is based on PMC, an asynchronous process algebra with multiple clocks. The keywords here are 'asynchrony' as the essential feature of distributed computation and the notion of a 'clock' as an elementary real-time mechanism. We base the discussion on an actual industrial product: The Brüel & Kjær 2145 Vehicle Signal Analyzer, an instrument for measuring and analyzing noise generated by cars and other machines with rotating objects. We present an extension of PMC by ML-style value passing and demonstrate its use on a simplified version of the Brüel & Kjær Signal Analyzer.

## 1 Introduction

The initial motivation for the work reported in this paper stems from an industrial case study pursued by the authors in the context of the CODESIGN project at the Department of Computer Science of the Technical University of Denmark, Lyngby. The task of this case study is the formal description and rational reconstruction of a commercial real-time measurement instrument, the Brüel & Kjær 2145 Vehicle Signal Analyzer [10]. Brüel & Kjær, an industrial partner associated with the CODESIGN project, is a big Danish manufacturer for measurement equipment and the 2145 is one of the most sophisticated of their products. The instrument — in its portable version — looks roughly as shown in Fig. 1. Its main purpose

*The first author has been supported by The Danish Technical Research Council and the second author by the Human Capital and Mobility Network EuroForm. Address of correspondence: Technical University of Denmark, Department of Computer Science, Building 344, DK-2800 Lyngby, Denmark, {hra,mvm}@id.dtu.dk



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

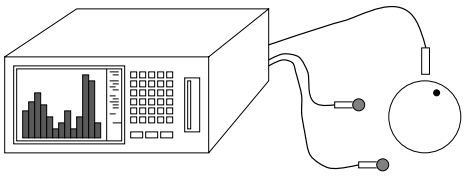


Figure 1: The Brüel & Kjær 2145 Vehicle Signal Analyzer

is to measure and analyze the noise produced by rotating mechanical objects such as car engines or turbines. It is applied in quality check and trouble shooting. As indicated in the figure basically two kinds of data are processed. The first is tacho information relating to the position, velocity, and acceleration of the rotating object. The second type of information is the sound produced, which is picked up by a number of microphones. The input signals are digitized and undergo fairly involved data processing to extract what is essentially frequency information, but linked up with the rotational data in one way or another.

When one studies the instrument's implementation one recognizes two salient features that must be accounted for by any attempt on a formal description of the instrument. The first is the fact that one is dealing with a truly distributed system, distributed both in terms of hardware as well as in terms of software. Depending on how one counts, one identifies at least four separate and dedicated hardware processors which are independently clocked and which communicate asynchronously. One of these processors runs a real-time operating system which in turn schedules three basic software functions in a quasi-parallel fashion.

The second insight one arrives at rather quickly is that it would be a hopeless undertaking to attempt a precise and complete specification of the instrument's internal timing behaviour. All one can reasonably expect is to capture a few and essential real-time aspects. But what are the essential real-time aspects? Of course, there is the obvious 'What-You-See-Is-What-You-Hear' response time constraint that says that the instrument must be fast enough for the test engineer to be able to relate the display output directly to the noise she or he is hearing. From the users point of view this is certainly a relevant real-time requirement. But there are more subtle and more important real-time constraints relating to the functional correctness of the measurement. In fact, when one talks to the engineers they insist that the main problem they are struggling with is to guarantee internal time consistency: to maintain the original exact time synchrony of the input data within the system, despite the fact that the signals are sampled independently and processed in a distributed fashion, despite the fact that the data split up into different submodules and reconverge later in yet another independently clocked subcomponent; and above all the



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

instrument must be able to measure absolute time with high precision in order to compute the current rotation speed, and relate it to the various signal data for later time-domain processing.

From this discussion we derive two central requirements for a prospective language to describe and program real-time systems such as the Brüel & Kjær 2145 Vehicle Signal Analyzer: Firstly, we are looking for an abstract approach that

- faithfully represents asynchronous and distributed computations.

In other words, our language must not, by illegitimate synchrony assumptions, mask out actual real-time synchronization problems in the implementation. Secondly, in order to master the complexity of the instrument the language

- must not mix up function and quantitative timing unnecessarily.

In other words, we must be able to focus on the essential real-time behaviour and purely functional aspects, and wherever appropriate ignore quantitative timing altogether.

In this paper we wish to put forward the real-time process language PMC [5] which has been conceived to comply with the two requirements above. It is in fact an extreme solution in the sense that in PMC all concurrent computations are asynchronous so that any global synchronization must be specified explicitly by the programmer. Also, PMC takes an extreme stand as regards the second requirement: it focuses on the qualitative aspects of real-time programming and does not attempt to capture quantitative timing, though this could be introduced as a derived concept.

PMC (*Processes with Multiple Clocks*) is an extension of Robin Milner's *Calculus of Communicating Systems* (CCS) by the notion of multiple clocks. Processes in PMC are described by their ability to communicate locally in a handshake fashion and synchronize globally on clocks. Clocks in this context are an elementary mechanism for achieving real-time constraints. They embody an abstract, qualitative, and local notion of time which can be interpreted as referring not only to real hardware clocks as in synchronous circuits, but also to time-out interrupts, global synchronization signals as in MODULA, the ticking of real process time, or the completion signal of a distributed initialization or termination protocol. PMC has a mathematical theory along the lines of CCS; the results obtained concern the formal calculus of PMC, its operational semantics, and complete equational axiomatizations for bisimulation equivalence and observation congruence [4, 3]. In this paper we extend PMC by value-passing using Standard ML [18] and illustrate its application as a programming language on a simplified version of the Brüel & Kjær 2145 Vehicle Signal Analyzer.

As mentioned before PMC is designed for describing truly distributed real-time systems with few but essential real-time constraints. This goal distinguishes it from



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

the usual approaches in the area.

On the one side, PMC does not build in any global synchrony assumption as in the real-time programming languages ESTEREL [7] and LUSTRE [13]. Global synchrony is implicit also in timed process algebras with the so-called *maximal progress property*, which essentially amounts to a globally synchronous, locally asynchronous model of computation. Examples are TPL [15] and TIMED CCS [26]. PMC, in contrast, can deal not only with globally synchronous, locally asynchronous behaviour but also with the more general class of globally asynchronous, locally synchronous behaviour. (A recent proposal for extending ESTEREL to achieve a similar effect can be found in [8].)

On the other side, whereas PMC concentrates on qualitative real-time constraints, the standard pattern of introducing time into process algebras aims at a precise and complete description of a real-time system's quantitative timing. Examples are ATP [22], TIMED CSP [25], BPA$\rho\delta$, and many others [23, 26, 21, 17, 16, 24]. These approaches use a global notion of time and describe the global real-time behaviour of the system quite precisely by inserting explicit delays. This may be necessary in many safety-critical applications, however, for real-time systems such as the Vehicle Signal Analyzer, it is overly realistic, for it implies that rather precise knowledge of the timing behaviour of the implementation is known or assumed; not only for the time-critical parts, but also for the remaining time-irrelevant aspects, which, so we believe, constitute the majority in practice. For instance, in a simple process like

$$P \;=\; a;\, b_1;\, \cdots b_n;\, P,$$

which performs an infinite sequence of $a$ actions separated by a sequence of $b_i$ actions, we might want to limit the time between any two $a$-actions without specifying anything about the intermediate $b_i$'s. The usual formalisms typically require a fixed delay or an interval of delays (as in [17]) to be assigned to each $b_i$, which means we are imposing unnecessary restrictions on them. In general, this will not be the most helpful solution as it might require almost clairvoyant skills: We must foresee the effects of our compiler and code optimization, have precise knowledge about the properties of our real-time operating system, and finally also of our hardware on which the program eventually is going to run.

# 2 PMC

In PMC concurrent systems are described by their ability to perform *actions* and synchronize with *clocks*. This dichotomy leads to a notion of transition system which distinguishes between pure action and pure clock transitions. One difference



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

between action and clock transitions is that actions embody local *handshake communication* whereas clocks embody global *broadcast synchronization*. Another is that action transitions are *nondeterministic* in general since they arise from parallel and distributed computations. Clock transitions, in contrast, are *deterministic* since they model the global passage of time. The idea that time passes deterministically is natural and appears to be common in timed process algebras, where it is known as the property of time determinism [23]. PMC was introduced in [5] and its mathematical theory was developed in [4, 3]. In this section we extend PMC by *value-passing* and ML-style *local declarations*, and present a simple operational semantics for *late binding* (see [20]).

As in value-passing CCS [19] we assume a set of process names *Proc*, channel names *Chan* and sets of values $\mathcal{V}$ and value variables *Var*. The semantics we present will be akin to symbolic transition systems [14]. We assume the existence of a *silent* action $\tau$ and take the set of *actions* to be $Act =_{\text{def}} \{c? \mid c \in Chan\} \cup \{c!v \mid c \in Chan, v \in \mathcal{V}\} \cup \{\tau\}$. Actions of the form $c?$ are *input actions* and $c!v$ are *output actions*. Note, input actions $c?$ do not carry a concrete value like output actions, they simply represent a commitment to communicate on channel $c$. This asymmetry between input and output captures the late binding semantics. Finally, in addition to the ordinary actions, PMC assumes a set of *clocks* *Clk* the elements of which are ranged over by $\sigma$.

The syntax of *value expressions* is taken from a subset of Standard ML – roughly the subset characterised by removing exceptions and references leaving us with a side-effect-free functional language. We will not describe this in detail, nor do we get involved with the type system for PMC and the semantics of value expressions. For the purpose of this paper it will be enough simply to refer to a (partial) evaluation relation for expressions. The syntax, type system, and evaluation semantics for expressions may be thought of as being taken over wholesale from Standard ML.

Process terms $t$ are generated by the following grammar:

$$
\begin{array}{rcl}
t & ::= & \texttt{stop} \\
& \mid & \alpha;\, t \\
& \mid & \texttt{if } e \texttt{ then } t_0 \texttt{ else } t_1 \\
& \mid & t_0 + t_1 \\
& \mid & t_0 \mid\mid t_1 \\
& \mid & \texttt{restrict } \mathit{cseq} \texttt{ to } t \\
& \mid & \texttt{timeout } t_0 \texttt{ on } \sigma \texttt{ as } t_1 \\
& \mid & t \texttt{ allowing } \sigma\mathit{seq} \\
& \mid & p(\mathit{eseq}) \\
& \mid & \texttt{let } d \texttt{ in } t \texttt{ end}
\end{array}
$$

Roughly, the meaning of the operators, in terms of their ability to perform actions



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

or to take part in clock ticks, is as follows. The process `stop` can do nothing, neither an action nor does it admit any clock to tick. The process $\alpha$; $t$ performs the prefix $\alpha$ and then behaves as $t$; it prevents all clocks from ticking, whence it is called 'insistent' prefix. The *prefix* $\alpha$ is either an input, an output or a silent prefix:

$$\alpha ::= c\ ?\ x \ \mid\ c\ !\ e \ \mid\ \tau.$$

The conditional process `if` $e$ `then` $t_0$ `else` $t_1$ behaves like $t_0$ or $t_1$ depending on the value of the (boolean) expression $e$. The process $t_0 + t_1$ behaves either as $t_0$ or $t_1$, the choice being made by the first action (but *not* by a clock-tick). The concurrent composition $t_0 \mid\mid t_1$ behaves like $t_0$ and $t_1$ executing concurrently, with possible communications. The process `restrict` *cseq* `to` $t$ behaves like $t$ but does not allow input and output actions on any of the channels in $cseq \in Chan^*$. Each one of the processes $t_0 + t_1$, $t_0 \mid\mid t_1$, and `restrict` *cseq* `to` $t$ takes part in a clock tick by having all of its components $t_0, t_1, t$ take part in it. Finally, `timeout` $t_0$ `on` $\sigma$ `as` $t_1$ behaves like $t_0$ if an initial action of $t_0$ is performed or a clock tick different from $\sigma$ occurs in $t_0$, however, if $\sigma$ occurs it behaves like $t_1$. This timeout operator is inspired by the timeout operator of Nicollin and Sifakis [22] which can be seen as a special case of ours where there is only one clock. The process $t$ `allowing` $\sigma seq$ behaves like $t$ but will take part in any tick from a clock in $\sigma seq \in Clk^*$ without changing state. Process constants can be instantiated as $p(eseq)$ by applying the process name $p$ to a sequence *eseq* of channel or clock names, or value expressions. The `let` construct introduces local *declarations* like in ML, *i.e.* `let` $d$ `in` $t$ `end` behaves like $t$ in an environment with the binding of identifiers to values, functions and processes as declared by $d$. We extend the declarations in ML to allow process declarations

$$\texttt{proc}\ p(aseq) = t,$$

where *aseq* is any sequence of channel or clock names, or value variables. Like in Standard ML we use the keyword `and` to connect mutually recursive declarations.

Two syntactic abbreviations will turn out to be useful:

$$\begin{array}{rcl} \texttt{await}\ \sigma;\ t & =_{\text{def}} & \texttt{timeout stop on}\ \sigma\ \texttt{as}\ t \\ \alpha\ \texttt{allowing}\ \sigma;\ t & =_{\text{def}} & \texttt{let proc}\ X = \texttt{timeout}\ \alpha;\ t\ \texttt{on}\ \sigma\ \texttt{as}\ X \\ & & \texttt{in}\ \ X \\ & & \texttt{end} \end{array}$$

The first process waits for the clock $\sigma$ to tick, whereupon it continues as $t$. The second process is a relaxed prefix, which admits clock $\sigma$ to tick freely until it performs action $\alpha$ whereupon it continues as $t$. The `let` construct applies a recursive definition with a fresh process name $X$, which must not occur free in $t$.

The semantics of PMC is given as a labelled *transition relation* $\rightarrow$. Labels are taken from the set $\mathcal{L} = Act \ \cup \ Clk$. Like in PMC without value-passing [5],



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

a transition with label $l \in \mathcal{L}$ is either a pure action transition, if $l \in Act$, or a pure clock transitions, if $l \in Clk$. The difference is that now actions carry value-passing information, and further that the transitions relates configurations instead of just process terms like in PMC. Configurations are introduced essentially to deal with local declarations `let` $d$ `in` $t$ `end`, *i.e.* with the situation where the processes of a term have different local environments. A *configuration* is either a pair $\langle D, t\rangle$ consisting of a sequence $D$ of declarations and a term $t$ (process or expression), or any of the process operators $op \in \{$`if_then_else_`, $+$, $||$, `restrict` $\vec{c}$ `to_`, `timeout_on` $\sigma$ `as_,_allowing` $\vec{\sigma}\}$ applied to configurations. For example, $\langle D_0, t_0\rangle \,||\, \langle D_1, t_1\rangle$ and `if` $\langle D, e\rangle$ `then` $\langle D_0, t_0\rangle$ `else` $\langle D_1, t_1\rangle$ are configurations. As usual a configuration will be closed if it contains no free identifiers. We denote the set of configurations by $\mathcal{C}$ and the set of closed configurations by $\mathcal{C}^{cl}$. A *declaration sequence* is a sequence of sets of mutually recursive declarations. In order to handle the late binding of values in input actions we use a special variable name # as a place holder. Let $\mathcal{C}^{\#}$ denote the set of configurations that has at most the free identifier #. Using this notation the transition relation $\rightarrow$ is a subset of $\mathcal{C}^{cl} \times \mathcal{L} \times \mathcal{C}^{\#}$.

We will need to assume that every well-formed syntactic declaration $d$ can be mapped to a sequence of sets of bindings by the map $\hat{}$ as indicated by the following example: If $d$ is

$$\begin{array}{l}\texttt{proc}\ p_1(\vec{x}_1) = t_1\\ \texttt{proc}\ p_2(\vec{x}_2) = t_2\\ \texttt{and}\ \ p_3(\vec{x}_3) = t_3\end{array}$$

then $\hat{d}$ is

$$\{p_2(\vec{x}_2) = t_2,\ p_3(\vec{x}_3) = t_3\}\{p_1(\vec{x}_1) = t_1\},$$

where a sequence is simply constructed by juxtapositioning the elements (using $\varepsilon$ for the empty sequence). Hence the first element of the above sequence contains the bindings for $p_2$ and $p_3$, the second and last element contains the binding for $p_1$. Note, in general a declaration sequence $D$ will also contain ordinary ML declarations for constants, functions, *etc.* but since we wish to focus on the PMC-related part, we shall not be bothered by how $\hat{}$ works on pure ML declarations.

For a declaration sequence $D$ we define the partial function of looking up and instantiating the process named $p$ with arguments $\vec{v}$, denoted $D(p)(\vec{v})$, by induction on the length of $D$: If $D = \varepsilon$, then $D(p)(\vec{v})$ is undefined, otherwise if $D = d\ D'$ we distinguish two cases; if $d = \{p_1(\vec{x}_1) = t_1, \ldots, p_k(\vec{x}_k) = t_k\}$ and $p = p_i$ for some $1 \leq i \leq k$, then $D(p)(\vec{v}) = \langle D, t_i[\vec{v}/\vec{x}_i]\rangle$; otherwise, if $p \neq p_i$ for all $1 \leq i \leq k$ or if $d$ is an ML declaration, then $D(p)(\vec{v}) = D'(p)(\vec{v})$. Hence, $D(p)(\vec{v})$ gives a configuration consisting of the body of $p$ where the arguments have been instantiated to $\vec{v}$ and a declaration sequence in which to execute the process.



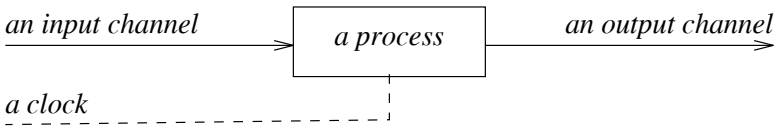
H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

Our operational semantics is parameterized in the ML evaluation relation $\Rightarrow$, where $\langle D, e\rangle \Rightarrow v$ means that in the environment of declaration sequence $D$, $e$ evaluates to $v$. Since expressions do not depend on processes the evaluation may safely ignore any process bindings in $D$. It will be convenient to extend this relation to channel and clock names by stipulating

$$\langle D, c\rangle \Rightarrow c \qquad \langle D, \sigma\rangle \Rightarrow \sigma.$$

The transition relation is given by the inductive set of rules shown in Fig. 2.

For the examples it will be useful to have some graphical representation of processes. To this end we introduce some informal terminology: the *input (output) sort* of a process is the set of channels on which a process inputs (outputs) values; the *clock sort* of a process is the set of clocks that a process is intended to be controlled by. It is customary to visualise sorts by means of flow-graphs:




# 3 A Signal Analyzer in PMC

We are now going to describe a simplified version of the Brüel & Kjær 2145 in PMC where we focus on some of the essential features of the actual instrument illustrating the use of clocks for the distributed programming of a real-time measurement problem. The main simplification consists in picking out only one measurement mode and trigger condition from the many possibilities available in the Brüel & Kjær 2145.

The simplified 2145 measures the noise produced by a large turbine in the run-up phase and at a certain critical rotation angle. The total result of the measurement shall be the peak value in three pre-defined frequency bands together with the velocities at which the peaks occurred. To solve our measurement problem we use the three basic components, Filter, Evaluation, Tacho, shown in Fig. 3. All three modules correspond to hardware components in the Brüel & Kjær 2145's implementation, and the formal description to follow is a (simplified) abstract view of the actual components' functionality.

The **filter** extracts the average energy of the incoming signal *sig* in a well-defined frequency band, and delivers the square root of this mean value on output *pwr*. There are two clocks associated with the filter characterizing its real-time behaviour. The first one, $\sigma_s$ is the sampling rate which determines the frequency resolution and the filter's maximal cut-off frequency. In the 2145 this is set at a fixed rate of $65kHz$.



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

$$\langle D,\ c\ ?\ x;\ t\rangle \xrightarrow{c?} \langle D,\ t[\#/x]\rangle \qquad \frac{\langle D,\ e\rangle \Rightarrow v}{\langle D,\ c\ !\ e;\ t\rangle \xrightarrow{c!v} \langle D,\ t\rangle}$$

$$\frac{B \Rightarrow \texttt{true} \quad C_0 \xrightarrow{l} C'}{\texttt{if } B \texttt{ then } C_0 \texttt{ else } C_1 \xrightarrow{l} C'} \qquad \frac{B \Rightarrow \texttt{false} \quad C_1 \xrightarrow{l} C'}{\texttt{if } B \texttt{ then } C_0 \texttt{ else } C_1 \xrightarrow{l} C'}$$

$$\frac{C_0 \xrightarrow{\alpha} C'}{C_0 + C_1 \xrightarrow{\alpha} C'} \qquad \frac{C_1 \xrightarrow{\alpha} C'}{C_0 + C_1 \xrightarrow{\alpha} C'} \qquad \frac{C_0 \xrightarrow{\sigma} C_0' \quad C_1 \xrightarrow{\sigma} C_1'}{C_0 + C_1 \xrightarrow{\sigma} C_0' + C_1'}$$

$$\frac{C_0 \xrightarrow{\alpha} C_0'}{C_0 \mid\mid C_1 \xrightarrow{\alpha} C_0' \mid\mid C_1} \qquad \frac{C_1 \xrightarrow{\alpha} C_1'}{C_0 \mid\mid C_1 \xrightarrow{\alpha} C_0 \mid\mid C_1'}$$

$$\frac{C_0 \xrightarrow{\sigma} C_0' \quad C_1 \xrightarrow{\sigma} C_1'}{C_0 \mid\mid C_1 \xrightarrow{\sigma} C_0' \mid\mid C_1'}$$

$$\frac{C_0 \xrightarrow{c?} \langle D_0', t_0'\rangle \qquad C_1 \xrightarrow{c!v} C_1'}{C_0 \mid\mid C_1 \xrightarrow{\tau} \langle D_0', t_0'[v/\#]\rangle \mid\mid C_1'} \qquad \frac{C_0 \xrightarrow{c!v} C_0' \qquad C_1 \xrightarrow{c?} \langle D_1', t_1'\rangle}{C_0 \mid\mid C_1 \xrightarrow{\tau} C_0' \mid\mid \langle D_1', t_1'[v/\#]\rangle}$$

$$\frac{C \xrightarrow{l} C'}{\texttt{restrict } \vec{c} \texttt{ to } C \xrightarrow{l} \texttt{restrict } \vec{c} \texttt{ to } C'}\ (l = c?, c!v \text{ implies } c \notin \vec{c})$$

$$\frac{C_0 \xrightarrow{l} C'}{\texttt{timeout } C_0 \texttt{ on } \sigma \texttt{ as } C_1 \xrightarrow{l} C'}\ (l \neq \sigma) \qquad \texttt{timeout } C_0 \texttt{ on } \sigma \texttt{ as } C_1 \xrightarrow{\sigma} C_1$$

$$\frac{C \xrightarrow{l} C'}{C \texttt{ allowing } \vec{\sigma} \xrightarrow{l} C' \texttt{ allowing } \vec{\sigma}}\ (l \notin \vec{\sigma}) \qquad C \texttt{ allowing } \vec{\sigma} \xrightarrow{\sigma_i} C \texttt{ allowing } \vec{\sigma}$$

$$\frac{C \xrightarrow{l} C'}{\langle D, p(\vec{e})\rangle \xrightarrow{l} C'}\ (\langle D, e_i\rangle \Rightarrow v_i,\ D(p)(\vec{v}) = C)$$

$$\frac{\langle \hat{d}\, D, t\rangle \xrightarrow{l} C}{\langle D, \texttt{let } d \texttt{ in } t \texttt{ end}\rangle \xrightarrow{l} C}$$

$$\frac{op(\langle D, t_1\rangle, \ldots, \langle D, t_n\rangle) \xrightarrow{l} C}{\langle D, op(t_1, \ldots, t_n)\rangle \xrightarrow{l} C}$$

Figure 2: Action and Clock Progress Rules. Recall that $\alpha$ ranges over actions, $\sigma$ over clocks, $l$ over both, and finally $op$ over $\{\texttt{if\_then\_else\_},\ +,\ ||,\ \texttt{restrict } \vec{c} \texttt{ to\_},\ \texttt{timeout\_on } \sigma \texttt{ as\_},\ \texttt{\_allowing } \vec{\sigma}\}$ in the last rule.



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

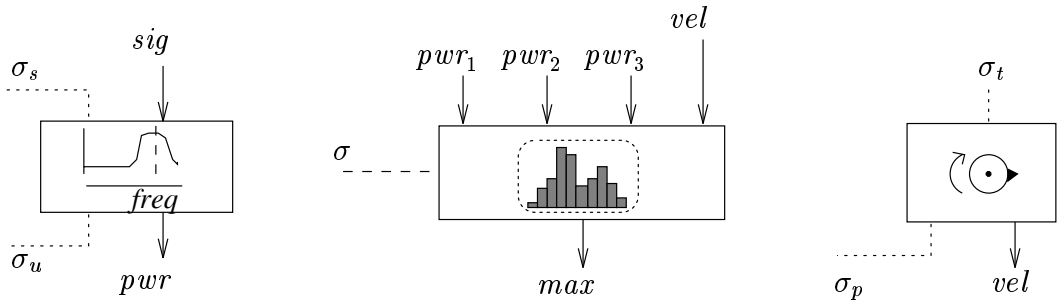



Figure 3: A Filter, Evaluation, and Tacho Component

The second clock, $\sigma_u$, is the update rate on the output side. It is the rate with which the accumulated averaged signal energy is updated on the output to be picked up and evaluated by the system. In general, $\sigma_u$ may be variable and smaller than the sampling rate depending on the speed of the successive computations or on how fast the frequency information of interest changes over time.

A description of the filter in PMC syntax reads as follows:

```
proc Filter(freq, sig, pwr, σs, σu) =
   restrict r to
      let
          proc F = await σs; sig ? x; r ? s; r ! filter(freq, x, s); F

          proc R(s,p,o) = timeout
                              timeout
                                    r ? s; R(s,p,o)
                                  + r ! s; R(s,p,o)
                                  + pwr ! o; R(s,p,o)
                              on σs as R(s,s,o)
                          on σu as R(s,p,p)
      in
          R(0,0,0) || F allowing σu
      end.
```

The unspecified function *filter*, parametrized by a frequency *freq*, a sample $x$, and a filter-state $s$, implements the filtering algorithm. The filter consists of two processes running in parallel and communicating on the internal channel $r$. The process $R(s,p,o)$ is a register with three state variables, $s$, $p$ and $o$. The first component can be set and read along the channel $r$. The last component holds the current value of the output line of the filter and it can always be read off by the output action $pwr\,!\,o$. At every tick of $\sigma_s$ the value of $s$ is copied to the second component,



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

and at every tick of $\sigma_u$ the value of $p$ is copied to the third component becoming the new output of the filter. The register is used by the process $F$ for storing the accumulated mean square of the signal energy. At the beginning of each iteration the process $F$ waits for the next tick of $\sigma_s$, reads in the new sample $x$ and retrieves the current value of $s$ from the register. From $x$ and $s$ it computes the new state $\mathit{filter}(\mathit{freq}, x, s)$ and updates the register.

The two-phase shifting of states in the register ensures that if a bank of filters is connected to the same $\sigma_s$ and $\sigma_u$, values read from the output lines of different filters between consecutive ticks of $\sigma_u$ will be consistent. *I.e.* they will be the result of computing the signal energy of the same number of samples. The reader is encouraged to try out a simplified version where the register only contains the state variables $s$ and $o$ and at every tick of $\sigma_u$ the value of $s$ is copied to $o$ while $\sigma_s$ is given free by `allowing`. With such filters unsynchronized values can occur: If some of the filters have performed the update of their registers and others not, the values read off are inconsistent.

The **tacho measurement** (the right-hand flow-graph in Fig. 3) computes the current rotation speed from the tacho pulse, which we may view as a variable clock $\sigma_p$. To get the velocity from this tacho clock we need to know the amount of time that has passed between any two pulses. This real-time information is implemented by another clock, $\sigma_t$, ticking off global system time. In the Brüel & Kjær 2145 this is done by a high-precision free-running timer oscillating at $1MHz$, yielding a $1\mu s$ time resolution. A description of the tacho as a PMC process is as follows:

```
proc Tacho(vel, σp, σt) =
   let
       proc T(c, e) = timeout
                        timeout
                          vel ! 1/e; T(c, e)
                        on σp as T(0, c)
                      on σt as T(c + 1, e)
   in
       T(0, ∞)
   end.
```

The state of the tacho $T(c, e)$ is specified by two parameters. The first one, $c$, counts the time between pulses, *i.e.* it is incremented with every $\sigma_t$ and reset with every $\sigma_p$ tick. The second parameter, $e$, holds the result count between two pulses; it is updated with $\sigma_p$. The current velocity, which is indirectly proportional to the result count can be read at any time with output action $\mathit{vel}\,!\,1/e$.

The last module to be specified is the **evaluation** module. A flow-graph for this module is found in Fig. 3. The task of the evaluation is to find the maximum peak



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

energies supplied at its inputs $pwr_i$, $i = 1, 2, 3$ in the run-up phase of the rotation. The run-up phase is a period of increasing velocity *vel*, beginning with a start value *start* and ending with a pre-defined stop value *stop*. The clock $\sigma$ serves to separate successive input vectors of synchronous frequency and velocity data. The evaluation module cycles through the states $E_{wait}$, $E_{comp}\ (m)$, and $E_{ready}$. In state $E_{ready}$ it is ready to start the next run-up measurement. When the velocity falls below the *start* margin it passes to state $E_{wait}$ where it waits for the velocity to enter the run-up interval $[start, stop]$. Then the actual computation state $E_{comp}\ (m)$ is entered. In this state the component reads in consecutive triples of frequency energies from $pwr_1, pwr_2, pwr_3$ and for each frequency channel memorizes the maximum value found so far along with the corresponding velocity. This computation is done on the state parameter $m$, a triple of pairs of maximal energies and corresponding speeds, using an appropriate ML function *max*. We use $m_0$ for the initial value of the state parameter. In concrete terms the PMC description of this process can be given as follows:

```
proc Eval(pwr1, pwr2, pwr3, max, vel, σ) =
   let
      proc E_wait       = await σ;
                          pwr1 ? p1; pwr2 ? p2; pwr3 ? p3; vel ? x;
                          if x < start then E_wait
                          else E_comp (max (m0, p1, p2, p3, x))

      and  E_comp (m)   = await σ;
                          pwr1 ? p1; pwr2 ? p2; pwr3 ? p3; vel ? x;
                          if x > stop then
                             max ! m allowing σ;
                             E_ready
                          else
                             E_comp (max (m, p1, p2, p3, x))

      and  E_ready      = await σ;
                          vel ? x;
                          if x < start then E_wait
                          else E_ready
   in
      E_ready
   end.
```

A few explanations are in order here. The fact that the sequence of input prefixes $pwr_1?p_1$; $pwr_2?p_2$; $pwr_3?p_3$; $vel?x$; blocks clocks is essential for it makes sure that no tick of $\sigma_u$ can intercept with the reading of the input lines, so that *Eval* obtains a



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

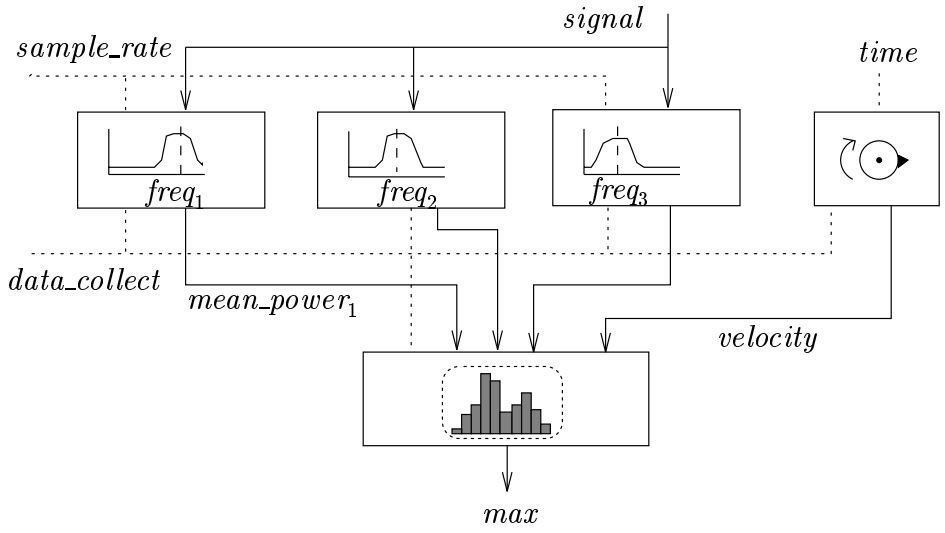



Figure 4: *Mini2145* – A Simple Version of the 2145 Signal Analyzer

time consistent view of the input. On the other hand, when the velocity has passed the upper margin, $x > stop$, we may safely allow the environment to run on freely until the results of the previous measurement have been picked up at output *max*. When this happens we prepare ourselves for a new measurement in state $E_{ready}$. This explains the relaxed prefix $max!m$ `allowing` $\sigma$; $E_{ready}$. The final observation made use of in the above formulation is that in state $E_{ready}$, where we wait for the velocity to fall below the start margin, we do not need to read in the frequency information, therefore the input action $vel?x$ suffices.

With the three components at hand we may now assemble our instrument as shown in Fig. 4. We take a bank of three filters each one tuned at a specific center frequency and have all filters sample the incoming sound signal by the same sampling rate. This ensures that all filters get a *consistent* view of the signal's shape. This is important as any imprecision in the synchronization of the sampling would result in a distortion of the measured results. Further, we connect the filters' output update rate with the tacho pulse, to obtain a vector of time-synchronous frequency energies and rotation speed relating to a *fixed position* of the rotating turbine. The evaluation module finally uses the velocity to pick out the frequency spectra corresponding to a predefined speed-interval in the run-up phase of the turbine. The PMC description of the overall system is now easily given:



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

```
proc Mini2145(signal, max, sample_rate, data_collect, time) =
  restrict mean_power_1, mean_power_2, mean_power_3, velocity to

    (  Filter(freq_1, signal, mean_power_1, sample_rate, data_collect)
    || Filter(freq_2, signal, mean_power_2, sample_rate, data_collect)
    || Filter(freq_3, signal, mean_power_3, sample_rate, data_collect)
    )  allowing time

    || Tacho(velocity, data_collect, time)
       allowing sample_rate

    || Eval(mean_power_1, mean_power_2, mean_power_3, max, velocity, data_collect)
       allowing sample_rate, time
```

Although this description contains no explicit timing constraints, it *does* contain all the information necessary to ensure proper functional real-time behaviour of the system. What remains is to decide on the realization and the speed of clocks. The Mini2145 features three clearly independent clocks modelling three different real-time aspects of the Brüel & Kjær 2145. Two of these clocks, the sampling rate and real time base are fixed rate, while the data collect rate is flexible. The point is that no matter how the three clocks are implemented all the constraints imposed on the system can be found in the above description. For instance, selecting the sampling rate to be a fixed clock running at $65k$Hz requires the Mini2145 be ready to synchronize on *sample_rate* at every 1/65000 second, which in turn requires the three filter processes to be able to each complete the treatment of one sample within this limit.

A more involved constraint occurs for the pulse detecting clock *data_collect*. Any external requirement given in the form of an acceptable range of pulse speeds (*e.g.* 0.01Hz–$30k$Hz) will require the Filters, the Tacho and the Eval process all to get ready to synchronize on *data_collect* when the pulse comes. Since the processes must communicate on various channels before this happens we are faced with constraints not only on the speed of actions internal to the processes but also on the communications between them.

# 4 Clocks and Real-Time Constraints

Given that the notion of 'clock' features prominently in our approach it is appropriate to be rigorous about our use of the term, and for that matter, about our view of real-time programming.



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

In fact, to get the right picture of our approach it is important to realize that the term 'clock' in its strict sense does not refer to the chronometer or an absolute notion of time but to the bell, *i.e.* the audible signal by which we tell the hour. The point we wish to make, of course, is that our use of clocks does not formalize the quantitative aspect of real time but rather the qualitative aspect of real-time, *viz.* that of a global synchronization event. There is indeed some risk of confusion as in the literature on timed semantics 'clocks' sometimes are used as a mechanism for measuring absolute quantitative time in order to time-stamp observations. Examples of such uses are the process algebra CIPA [1] and the timed automata of Alur and Dill [2].

Although, at first glance our approach is somewhat akin to having a discrete time-domain, *viz.* using a single clock to tick off intervals of a global and absolute time, the intended interpretation here is more abstract: In general, PMC processes would use a set of unrelated clocks which *a priori* proceed independently. As mentioned in the beginning, in any actual implementation these clocks may have a variety of different realizations: They could be chosen to be real hardware clocks running at fixed speed, or more relaxed clocks with an allowed range of time-intervals between successive ticks. The fixed clocks *sample_rate* and *time* in the Brüel & Kjær 2145 are examples of the first kind, whereas as the pulse *data_collect* is an example of the second kind. However, some clocks may even run entirely independent while others are derived multiples of a distinguished master clock. But not only may the hardware interpretation apply, also software realizations are adequate: a clock may represent a time-out interrupt, a global synchronization signal, or the completion signal of a distributed initialization or termination protocol.

When we say that clocks are a primitive real-time mechanism then we do suggest that they capture certain properties of real time. There is, however, one crucial property not captured by clocks, and this is the ceaseless progress of time. Real time, as it is usually perceived, is an independent physical parameter that cannot be prevented from continuously proceeding towards infinity. This progress of time cannot be modelled by clocks. A clock in PMC is an internal signal which all components of a system are free to block or synchronize on. In other words, a process may produce a *time-lock* preventing a particular clock from ticking ever again. In PMC time-locks indicate the violation of a real-time constraint. If for example the Mini2145 is put in parallel with a process that occassionally gets into a state where it stops sending new samples on the channel *signal*, the filters will stop the clocks *sample_rate* and *data_collect* indefinitely. Another example occurs in synchronous circuits where a time-lock is produced by feed-back loops that do not contain a clocked register [4].



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95),
pp.620-635, Springer 1995 (LNCS 915).

# 5 Conclusion and Future Work

The ideas put forward in this paper aim at a qualitative approach to real-time programming that focuses on functional correctness and factors out issues like response time, measurement resolution, and calibration. The approach, which is based on PMC and emphasizes the importance of clocks, was illustrated on the Brüel & Kjær 2145 Vehicle Signal Analyzer.

It is worth to be stressed that we do not propose to ignore quantitative timing altogether. As a matter of fact, in our example analyzer we do have, implicitly, constraints on the implementation of clocks. For instance, the time base clock must be a high-precision fixed-frequency oscillator, for otherwise, the actual rotation speed cannot be computed correctly. Also, the sampling rate must be higher than the update rate, *etc.* Clearly, nothing prevents us from specifying timing properties initially as requirements on the clocks and actions of a design but – so is our thesis – ultimately their satisfaction cannot be determined until the final implementation is developed. For instance, determining the actual frequency of the Mini2145's time-base clock and its precision is an issue of calibration not of programming.

Thus, the approach we follow with PMC is to provide a powerful, high-level operational description language for which satisfaction of timing constraints will be determined from the final machine-executable code. It is our hope that by being very careful in the way the compilation is carried out, we shall be able to lift this information to a higher-level to guide the design by providing analysis information. For instance, by compiling parts of the description and estimating the execution time of this partial code information on clocks may be obtained. Hence, the emphasis is on providing information to the programmer and not to require him to perform detailed calculations on timing requirements. Of course, timed automata [2] and temporal logics such as the Duration Calculus [11] are good candidates for expressing timing requirements but we do not want this information to enter the process description.

The timing of code ultimately depends on the choice of the target machine(s); any attempt to estimate the execution times early in the design must rely on a very carefully designed compilation strategy. We believe that any such strategy should be based on a clear operational semantics of the language that reveals in detail the steps that have to be performed and where choices must be made.

For expressing dynamic behaviour PMC uses the basic constructions of Milner's CCS and for computations on values fragments of Standard ML. Of course, there is a tension between having a rich language and being able to derive real-time faithful implementations. We handle this by allowing a rich language that can be useful for initial high-level descriptions and to run simulations, and only give time-respecting



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

implementations for some reasonable subsets of the language – any future advances in compilation technology could then extend these subsets. The design goal in such a framework is to refine a high-level description into one within one of the executable subsets. All this takes place within the *same* language, which makes possible the co-existence and debugging of descriptions containing both low-level and high-level components.

Currently, a prototype implementation for PMC is under development, using the ML Kit of Birkedal *et.al.* [9]. A simulator and prototype compilers for mono-processor and multi-processor architectures are planned. The Brüel & Kjær 2145 will be the major test example.

# References

[1] L. Aceto and D. Murphy. On the ill-timed but well-caused. In E. Best, editor, *Proc. Concur'93*, pages 97–111. Springer LNCS 715, 1993.

[2] R. Alur and D. Dill. The theory of timed automata. In de Bakker et al. [12], pages 45–73.

[3] H. R. Andersen and M. Mendler. A complete axiomatization of observation congruence in PMC. Technical Report ID-TR:1993-126, Department of Computer Science, Technical University of Denmark, December 1993.

[4] H. R. Andersen and M. Mendler. A process algebra with multiple clocks. Technical Report ID-TR:1993-122, Department of Computer Science, Technical University of Denmark, August 1993.

[5] H. R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In D. Sannella, editor, *Programming Languages and Systems – ESOP'94*, pages 58–73. Springer, LNCS 788, 1994.

[6] J.C.M. Baeten and J.W. Klop, editors. *Proceedings of CONCUR '90*, volume 458 of *LNCS*. Springer-Verlag, 1990.

[7] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer LNCS 197, 1984.

[8] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating reactive processes. In *Principles of Programming Languages POPL'93*, pages 85–98. ACM, 1993.

[9] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit, Version 1. Technical Report, DIKU, March 1993.

[10] Brüel & Kjær. *Vehicle Signal Analyzer Type 2145, User Manual Vol. 1*, April 1994.



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).

[11] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[12] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *LNCS*. Springer-Verlag, 1991.

[13] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.-C. Glory. Specifying, programming and verifying real-time systems using a synchronous declarative language. In *Workshop on automatic verification methods for finite state systems*, Grenoble, France, June 12–14 1989. Springer LNCS 407.

[14] M. Hennessy and H. Lin. Symbolic bisimulations. Technical Report 1/92, University of Sussex, April 1992.

[15] M. Hennessy and T. Regan. A process algebra for timed systems. Computer Science Technical Report 91:05, Department of Computer Science, University of Sussex, April 1991. To appear in Information and Computation.

[16] Jozef Hooman. *Specification and Compositional Verification of Real-Time Systems.* Number 558 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[17] Chen Liang. An interleaving model for real-time systems. Technical Report ECS-LFCS-91-184, Laboratory for Foundations of Computer Science, University of Edinburgh, November 1991.

[18] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT press, 1990.

[19] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[20] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. Technical Report SICS/R–91/03–SE, Swedish Institute of Computer Science, 1991.

[21] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In Baeten and Klop [6], pages 401–415.

[22] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: theory and application. Technical Report RT-C26, LGI-IMAG, Grenoble, France, December 1990.

[23] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In de Bakker et al. [12], pages 526–548.

[24] G. Reed and A. Roscoe. A timed model for communicating sequential processes. In Laurent Kott, editor, *Proceedings of the 13'th ICALP*, pages 314–323. Springer, LNCS 226, 1986.

[25] S. Schneider, J. Davies, D.M. Jackson, G.M. Reed, J.N. Reed, and A.W. Roscoe. Timed CSP: Theory and practice. In de Bakker et al. [12], pages 526–548.

[26] Yi Wang. Real-time behaviour of asynchronous agents. In Baeten and Klop [6].



H. R. Andersen, M. Mendler: Describing a signal analyzer in the process algebra PMC -- A case study.
In P. D. Mosses, M. Nielsen, M. Schwartzbach (eds.), Theory and Practice of Software Development (TAPSOFT'95), pp.620-635, Springer 1995 (LNCS 915).