A programming-language extension for distributed real-time systems

A Programming-Language Extension for Distributed Real-Time Systems

by

J. Hooman and O. van Roosmalen

97/02

ISSN 0926-4515

Computing Science Reports 97/02
Eindhoven, January 1997

A Programming-Language Extension for Distributed Real-Time Systems

Jozef Hooman and Onno van Roosmalen

Dept. of Computing Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl and wsinonno@win.tue.nl

November 25, 1996

Abstract

In this paper we propose a method for extending programming languages that enables the specification of timing properties of systems. The way time is treated is not language specific and the extension can therefore be included in many existing programming languages. The presented method includes a view on the system development process. An essential feature is that it enables the construction of (hard) real-time programs that may be proven correct independently of the properties of the machines that are used for their execution. It therefore provides a similar abstraction from the execution platform as is normal for non-real-time languages. The aim of this paper is to illustrate the method and demonstrate its applicability to actual real-time problems. To this end we define a simple programming language that includes the timing extension. We present a formal semantics for a characteristic part of the language constructs and apply formal methods to prove the correctness of a small example program. We consider in detail a larger example, namely the mine-pump problem known from the literature. We construct a real-time program for this problem and describe various ways to map the program to an implementation for different platforms.

1 Introduction

Including time in programming languages in a proper way, i.e. a way that is syntactically concise, semantically simple and that allows formal verification of programmed timing behavior, is a longstanding issue. If one considers current practice in programming of real-time systems and the languages that most frequently are being used there (e.g. Ada, C), it seems that this issue has not yet been satisfactorily resolved. This is in spite of the fact that much research has been devoted to this subject and at present a number of proposals for language constructs expressing real-time constraints exist, particularly in the realm of object-oriented programming languages and methods.

The interest in the combination of object orientation and real time stems from the current popularity of object-oriented languages in non-real-time programming and should not be taken as a sign that a proper approach has already been found for non object-oriented languages. We think that that is not the case and that the lack of a sound basis for real-time programming causes programming languages to evolve without a co-evolution in possible real-time extensions. Object orientation places particular emphasis on composability and reusability of program components. These principles are also important in the construction of real-time software and must be taken into consideration in the search for proper real-time constructs. However, apart from that, object orientation and real-time issues are largely independent and should be considered separately.

The real-time programming approach presented in this paper has first been introduced in the object-oriented programming language *Deal* that was developed in the context of the Dependable Distributed Operating System (Dedos) project at Eindhoven University [13]. In the present paper we do not use Deal. Instead we introduce a simple concurrent language that is augmented with timing constructs. We do this for the following reasons: (1) the timing extensions that we will discuss are more generally applicable, and (2) we want to provide insight in the basic ideas underpinning our approach to real-time programming without the danger of creating confusion with typical object-oriented issues and (3) it is easier to give a formal semantics for this simple language. Object-oriented languages are in general more complex and some problems with making such languages concurrent, such as inheritance anomalies [21], are still being discussed. A well-defined semantics is needed in then present context to be able to demonstrate that correctness of real-time programs can be established independent of the execution platform.

2 Composability and reusability of real-time programs

It is generally recognized that problem decomposition and stepwise refinement [25, 9] are important programming strategies. When only functional aspects of programs are considered, there are no major problems in applying these strategies: the functional behavior of a component can be made to depend strictly on the explicitly declared interfaces offered by others. Thus, based on the interface properties, functionality can be combined to form more complex behavior. This composability property is essential to enable software reuse and the efficient construction of complex systems. In contrast, if one considers timing behavior of a program component, implementation details of other, concurrently executing components may become important. e.g. in a multitasking system the time required to complete a certain computation is sensitive to claims on the processor(s) made by other program parts that solve completely independent concerns. We will refer to these other program parts as the *context*. The hidden coupling, i.e. coupling that is not explicitly on the interface between components, is called *context dependence* here. Thus, timing aspects of a program or program part, in particular execution durations, may be context dependent. In addition timing behavior usually depends on the execution platform. A platform is characterized by the processor types, their configuration, and the operating system with its execution protocol.

Non-real-time programming approaches provide a context and platform independent way of describing algorithms. That is, the design decisions and the resulting program are not influenced by details of context and execution platform but solely depend on the specification of the system or subsystem that is to be constructed. This abstraction from platform and context, which is so typical in non-real-time, is also desired for real-time problems. It is a prerequisite for composability and reusability.

We distinguish two types of context and platform dependencies: (1) those that can be explicitly identified in the program text (2) those that implicitly influence program design.

An example of first type are hard coded process priorities. Priorities are selected relative to other, functionally independent processes (context), and also depend on the employed scheduling regime (platform). Another example is the selection of commensurable periods of control loops that are functionally independent but are included to obtain a program that is schedulable off line. If independent program parts are combined in other ways, frequencies must be reconsidered and possibly changed. Also, if one considers the present practice in distributed (fault-tolerant) algorithms, which often also must satisfy timing constraints, one observes that the solutions are usually very much tuned to a particular property of the platform (such as lock step synchrony). To understand an algorithm or to prove its correctness the platform properties must be made explicit and described separately. It is not sufficient to consider just the algorithm to prove that the specification is satisfied.

An example of implicit dependencies (e.g. dependencies introduced through a programming approach) is the realization of timing constraint of a program by introducing additional constraints (i.e. constraints that do not follow directly from the specification) on execution durations of sub-programs. The latter constraints can often only be sensibly chosen if the programmer has some information available about the platform and such constraints will restrict the number of platforms or platform configurations that can properly execute the program.

3 Platform-independent real-time programming

The real-time programming approach described in this paper offers a solution to the previously mentioned problems. In particular, the approach has the following properties:

- Timing and functional aspects are integrated in one language;
- Program semantics is context and platform independent;
- It yields composability and enables abstraction and stepwise refinement of timing behavior;
- Correctness of a program or a program part can be formally established.

To obtain these properties we distinguish two phases in real-time software development: (1) a completely platform independent programming phase, and (2) a system generation phase where all platform and context dependencies are addressed. The platform independence in the programming phase is achieved by extending the programming language with timing annotations that enable the specification of timing behavior without describing the implementation. With the timing annotation it is possible to limit the programmed timing constraints to those ones that are implied by the problem specification. No additional constraints need to be programmed to "implement" the timing specification.

In the system generation phase it must be established that the program can be realized on the selected platform, i.e. that the platform is powerful enough to satisfy the timing constraints that are expressed in the program. In addition mechanisms must be offered by the platform to implement the expressed behavior. This usually comes down to scheduling the application appropriately. The main difference between such a system generation phase and the compilation step in non-real-time system development is the possibility for the system generator to conclude that no implementation can be found for the selected platform, (i.e. a feasible schedule cannot be found).

Thus, the listed properties are obtained through a number of measures on three different but interdependent aspects of software development:

1. the development strategy: the strict separation of concerns during the programming phase and the platform dependent implementation phase,

2. the programming style: prescriptions on how the timing annotation should be employed, for instance avoiding the introduction of constraints that are not implied by the specification,

3. the programming language: i.e. the syntax of the timing extensions.

It is important to note that software components can be programmed independently. Their behavior, including the timing behavior, is constrained only by the specification of the problem or sub-problem they address. Verifying system behavior is split into two distinct activities corresponding to the two phases: (1) Establishing program correctness against the functional as well as the timing specification, possibly separately for each component. This provides a result independent of the context and the platform. (2) Establishing the existence of a feasible and correct schedule (i.e. one that satisfies the constraints expressed in the program) to execute the system components in composition on a specified platform. If step (2) fails the software designer can change the platform or change the program. Naturally, the change in the program should be confined to using more efficient algorithms (with lower computational complexity), or finer grained concurrency, in a way that keeps platform considerations out of the program design. (Concurrency optimizations can also be done automatically during system generation, thus avoiding the danger of introducing concurrency that is not strictly inspired by the specification of the real-time problem that is being solved [24].) The hope is that the envisaged system generation can be automated to a large extent. Real-time system-development tool kits that are presently available justify such hope.

The main purpose of this paper is to illustrate our approach by an example, show its practical implications and argue for its feasibility. For these purposes we introduce a simple programming language and use a simple example application known from the literature, the "mine pump" system. First we describe the language and its semantics, motivate the language design choices and show how verification of programs can be carried out (sections 4 to 5). Next we give an informal specification of the mine pump problem (sections 6 to 7). We provide a program that constitutes a solution to this problem (sections 8 to 9), and describe how it can be implemented on various platforms through a system generation step (sections 10 to 12). After that we will discuss our approach and compare it with various other proposals for including timing constraints in programming languages (section 13 to 14).

4 A simple parallel programming language

Gligor and Luckenbaugh [10] have published general requirements that should be met by methods for the construction of real-time systems. They divided these requirements into four groups, two of which are of particular interest to real-time and two are more general:

1. Control requirements

2. Timing requirements

3. Distributed systems requirements

4. "Good programming" requirements

Because we focus on constructs for expressing timing constraints only item (2) and (4) are relevant in the present context.

Concerning item (2) Gligor and Luckenbaugh do not say more about language constructs, than that they must enable the programmer to specify desired response time and specify strict upper bounds on the number of executions of all loops and program paths. Indeed, the most common requirements on real-time systems, which are at the same time most difficult to guarantee, are hard deadlines for programmed responses on external stimuli. However, in addition to deadlines, there are other timing requirements on systems which are less difficult to guarantee, but must also be taken into account when writing a program, e.g. the requirement that a particular response should take place *after* a certain time. A real-time programming language should deal with all timing aspects in a syntactically uniform and concise manner. This latter requirement falls in category (4), for which [10] describes the following requirements:

- The number of language primitives should be kept small.
- The primitives should be orthogonal (i.e., two or more primitives should not duplicate functionality).
- The principle of "separation of concerns" should be followed in designing the language primitives (e.g. , avoid primitives like the *Monitor* construct which combines synchronization, communication and data abstraction).[1]
- Each primitive should have a simple semantics definable by Hoare-like axioms or Mills' functions.
- Each primitive and each statement must have a formal definition.

In section 13 we will evaluate our language extension with respect to these requirements.

4.1 Syntax

Our starting point is a normal imperative concurrent programming language with asynchronous message passing along channels. We add an annotation to statements which contains timing information expressed using timing variables. We also introduce special device variables which enable communication with devices. Let *VAR* be a nonempty set of program variables, *RTVAR* be a nonempty set of timing variables, *DEV* be a nonempty set of device variables, *CHAN* be a nonempty set of channel names, and *CONST* be a domain of constants. The variables in *VAR* are all local (no global variables exist in this language).

The syntax of our programming language is given in table 1, with $n \in \mathbb{N}$, $x, x_1, \ldots, x_n \in VAR$, $m, m_1, \ldots, m_n \in RTVAR$, $d, d_1, \ldots, d_n \in DEV$, $c, c_1, \ldots, c_n \in CHAN$, $\mu \in CONST$, and R a binary relation symbol (such as $=$, $<$, $\geq$, etc).

Value Expression	$e ::=$	$\mu \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::=$	$e_1 = e_2 \mid e_1 < e_2 \mid$ **not** $b \mid b_1$ **or** $b_2 \mid b_1$ **and** b_2
Moment Expression	$me ::=$	$\mu \mid m \mid (me) \mid me_1 + me_2 \mid me_1 - me_2 \mid me_1 \times me_2$
Timing Annotation	$TA ::=$	$m := me \mid ?m \mid R\,me \mid TA_1 \,;\, TA_2$
Primitive Statement	$PS ::=$	**skip** $\mid x := e \mid in(d, x) \mid out(d, e)$
Statement	$S ::=$	$PS \mid PS[TA] \mid$
		$send(c, e, [\,!me\,]) \mid receive(c, x, [\,?m\,]) \mid flush(c) \mid$
		if b **then** S_1 **else** S_2 **fi** $\mid$ **while** b **do** S **od** $\mid$
		select $receive(c_1, x_1, [\,?m_1\,])$ **do** S_1 **od**
		or $receive(c_2, x_2, [\,?m_2\,])$ **do** S_2 **od**
		...
		or $receive(c_n, x_n, [\,?m_n\,])$ **do** S_n **od**
		endselect $\mid$
		$S_1 \,;\, S_2$
Process	$P ::=$	$S \mid P_1 \parallel P_2$

Table 1: Syntax of the Parallel Programming Language

[1] Interestingly enough, exactly this kind of mixing of concerns is what results in the phenomenon of concurrency anomalies in object oriented languages.

4.2 Informal semantics

For programs without timing annotations we use the conventional untimed semantics in which no assumptions are made about the speed of actions. Timing annotations are used to restrict the set of possible behaviors and they can be seen as constraints on the implementation/scheduler. Since there is no notion of platform in the programming phase, the definition of time units is provided by the problem domain. Therefore, we describe the timing behavior of a program from the viewpoint of an external observer with his own clock. Thus, although parallel components of a system might have their own, physical, local clock, the observable behavior of a system is described in terms of a single, conceptual, global clock. Since this global notion of time is not incorporated in the distributed system itself, it does not impose any synchronization upon processes. In this paper we use the non-negative reals as our (dense) time domain: $TIME = \{\tau \in \mathbb{R} \mid \tau \geq 0\}$. All times are taken relative to the start of the program, taken at time 0.

Informally, the statements of our programming language have the following meaning, using $\equiv$ to denote syntactic equality.

- **skip** has no effect, i.e. does not change any of the variables.
- Assignment $x := e$ assigns the value of expression e to the variable x.
- $in(d, x)$ denotes reading a device variable (register) d and assigning the obtained value to the variable x.
- $out(d, e)$ denotes writing the value of expression e into the device register d.
- To describe the meaning of an annotated primitive statement $PS[\,TA\,]$ we first define the *execution moment* of a primitive statement PS, denoted by $em(PS)$. The execution moment of PS is a point in time between (or possibly at) the start and the termination time of PS. It usually corresponds to the moment at which the state change that is the result of the statement is effectuated. $PS[\,TA\,]$ has the following meaning:
  - If $TA \equiv m := me$ then the value of the moment expression me is assigned to m. This is called a timing assignment.
  - If $TA \equiv ?m$ then the execution moment of statement PS, $em(PS)$, is assigned to m. This is called a time measurement.
  - Henceforth we identify relation symbol R and its interpretation.
    If $TA \equiv R\,me$ then we have $em(PS)\;R\,me$. This is called a timing requirement.
  - if $TA \equiv TA_1\;;\;TA_2$ then we have the sequentially combined effect of TA_1 and TA_2. (If this leads to $false$ the timing constraint can never be satisfied and an implementation of the program cannot be constructed. For instance **skip**$[\;< 5; > 5\,]$ has an unsatisfiable constraint.)

  Thus, we have three types of statements in the annotation: (1) timing assignments, that enable the manipulation of timing variables (2) time measurements which can be used to record the execution moment of the annotated program statement, (3) timing requirements, which expresses a timing constraint on the execution moment of the annotated statement.
- $send(c, e, [\,!me\,])$ denotes an asynchronous send action, that is, the value e is transmitted along channel c. Further, moment expression me is communicated to the receiver (conceptually, i.e., this need not be implemented). We assume that messages are stored in an infinite FIFO buffer.
- $receive(c, x, [\,?m\,])$ denotes a receive action which receives a value from channel c, namely the first available message in the FIFO buffer of this channel, and assigns this value to x. Further a timing value is received (conceptually) and assigned to m. This statements waits until a message is available.

- *flush*(c) denotes an action by which all messages in the buffer of channel c are removed and discarded.
- **if** b **then** S_1 **else** S_2 **fi** denotes the usual conditional choice construct.
- **while** b **do** S **od**, the traditional while statement.
- **select** *receive*($c_1, x_1, [\,?m_1\,]$) **do** S_1 **od**
  or *receive*($c_2, x_2, [\,?m_2\,]$) **do** S_2 **od**
  ...
  or *receive*($c_n, x_n, [\,?m_n\,]$) **do** S_n **od**
  endselect
  is a select statement. This statement waits until a message is available on one of the channels $c_1, \ldots, c_n$. Then a channel on which a message is available is selected, the message is received, and the corresponding statement is executed. The selection of the channel along which a message is received is non-deterministic, with the restriction that it should be fair. That is, if there is a message in the FIFO queue of a channel, then this message should be received eventually.
- S_1 ; S_2 indicates sequential composition.
- $P_1 \parallel P_2$ denotes parallel composition of processes P_1 and P_2.

We use **if** b **then** S **fi** as an abbreviation of **if** b **then** S **else skip fi** and $[\,TA\,]$ as an abbreviation of **skip**$[\,TA\,]$. For a send action we use *send*(c, e), *send*($c, [\,!me\,]$), and *send*(c) if moment expression and/or value expression are irrelevant. Similarly, we use *receive*(c, x), *receive*($c, [\,?m\,]$), and *receive*(c). In the following we will treat the constant *true* as a boolean expression, so that we can write **while** *true* **do** , to denote a non-terminating repetition.

Given a particular program, all execution traces are allowed that satisfy the specified timing constraints. Also we assume progress and finite variability, i.e. each message will ultimately arrive and any finite amount of progress in the program's execution will happen within a finite amount of time. Apart from this nothing is known about progress of statement execution or time required for the delivery of messages. If a program has passed a feasibility test for the execution platform at hand, which implies that the platform can implement the program, it is guaranteed that the execution mechanism satisfies the semantics of the program including the assumed liveness property.

To solve a real-time programming problem, it is normally not necessary for the programmer to fix or consider the execution moment or duration of every statement in the program. Almost always requirements only exist for the execution moment of statements that are related to some (observable) external event. Hence, the timing annotations, usually, only refer to execution moments of a few relevant statements. There is no need to explicitly consider the execution duration of all statements in between: this can be deferred to the system generation phase.

4.3 Syntactic restrictions

Programs should satisfy the following syntactic restrictions

- Channels connect two processes, that is, for a given channel c at most one parallel process may contain statements of the form *send*($c, e, [\,!me\,]$) and at most one other process may contain statements of the form *receive*($c, x, [\,?m\,]$).
- Processes cannot share variables, i.e., for a (sub)program $P_1 \parallel P_2$ a variable x may not occur in both P_1 and P_2. We sometimes use variables with the same name in different processes. It is always assumed that such variables can be distinguished for instance by labeling them with the process name in which they are used. In the following we will list variable names for each process. Constants are global and their names may appear anywhere in the program, they are given by the set $const \subset CONST$. The variables used in a program P are given by the sets $var(P) \subset VAR$, $rtvar(P) \subset RTVAR$ and $dev(P) \subset DEV$.

- Expressions in the annotations are syntactically isolated from the rest of the program by using square brackets. We distinguish time domain variables (variables used in between square brackets, *RTVAR*) and program domain variables (variables used outside the brackets: the other variable sets). The scope of variables is restricted to one domain only.

The reasons for the last syntactic restriction are twofold. First, separability of real-time constraints and functional aspects is achieved this way. Second, it becomes syntactically impossible to introduce data dependencies in the timing requirements, i.e. values of variables in the program domain cannot be used to formulate timing constraints. Only time measurements, i.e. observations of the execution moments of statements, yield new values for time domain variables that can be used in timing constraints. This feature enables the static (off-line) analysis of timing constraints which is required to prove correctness and to construct off-line schedules that satisfy the constraints.

5 Formal verification of time-annotated programs

In a couple of steps, we illustrate how formal verification of time annotated programs can be carried out. We present a formal, axiomatic, semantics of (part of) the language by formulating proof system which express when a program satisfies a certain specification. First the specifications are introduced, using an extended and modified version of Hoare triples (program, precondition, postcondition), similar to [14]. Next the proof system is given by formulation a set of compositional rules and axioms. Compositionality implies that one can reason with the specifications of components without knowing their program text. Program verification is illustrated by a small example of a water level control program. We give the top-level specification of this example and refine it towards an implementation. The correctness proof of the resulting program will be outlined. Details of the proof are given in a appendix A. This example will be relevant in the full mine-pump problem that will be considered in the next sections. A number of program design decisions that will be taken there can be better understood after this section.

5.1 Specifications

Specifications have the form $\langle\langle A \wedge term \rangle\rangle\ S\ \langle\langle C \rangle\rangle$ where A is an assertion called *assumption*, S a program, and C an assertion called *commitment*. In the assertions we use program variables, such as x, y, etc., and two special variables *em* and *term*. These terms are only used for local reasoning of a sequential program and are not part of the external interface of a component. They have the following meaning.

- In A, x represents the value of program variable x in the state before the execution of S. If S terminates, then in C it represents the value of x in the final state of S.
- *em* denotes the *execution moment*; in A the execution moment of the statement preceding S (0 if there is no such statement) and in C the execution moment of the last executed primitive statement of S.
- *term* is a boolean variable denoting termination; in A termination of the statement preceding S (*true* if there is no such statement) and in C it denotes whether S terminates.

In addition to these terms for sequential reasoning, components usually have an external interface, that is, a set of externally observable events. Since we deal with real-time systems, the timing of these events is important. Further we can introduce axioms to express the assumptions about the relation between sending and receiving. A description of these axioms is not needed for the example that we present and is outside the scope of the current paper.

In addition to message passing, the programming language offers device registers for components to communicate with their environment. A device register can either be set by the environment or be set by the process, but not both. In the former case the state of the environment will be reflected in the value obtained by a read action, in the latter the program will determine the state of the device. The following primitives are added to the assertion language.

- $read(d, v)$ **at** t to express that value v is read from d at time t, and
- $set(d, v)$ **at** t to express that d is set to value v at time t.

To abstract from read or written values, two abbreviations are introduced.

- $read(d)$ **at** $t \equiv \exists v : read(d, v)$ **at** t
- $set(d)$ **at** $t \equiv \exists v : set(d, v)$ **at** t

Further we introduce the notation

- $d(t)$ to represent the value of register d at time t.

Henceforth, frequently $(d = v)$ **at** t is used as another notation for $d(t) = v$. For a predicate P **at** t and a set (usually an interval) $I \subseteq TIME$, we use the abbreviations

- P **during** $I \equiv \forall t \in I : P$ **at** t,
- P **in** $I \equiv \exists t \in I : P$ **at** t.

The next axiom expresses that the value read corresponds to the actual value of of the device register, d:

$$read(d, v) \textbf{ at } t \rightarrow d(t) = v \qquad \text{(READAX)}$$

Further, if a process writes a device register d, the value of d equals the last written value (or the initial value at 0), that is,

$$\begin{aligned} d(t) = v \leftrightarrow\ & (\exists t_0 \leq t : set(d, v) \textbf{ at } t_0 \wedge \neg set(d) \textbf{ during } (t_0, t]) \vee \\ & (v = d(0) \wedge \neg set(d) \textbf{ during } [0, t]) \end{aligned} \qquad \text{(SETAX)}$$

We postulate that at most one value is written at any point in time.

$$set(d, v_1) \textbf{ at } t \wedge set(d, v_2) \textbf{ at } t \rightarrow v_1 = v_2 \qquad \text{(UNIQUEAX)}$$

Finally, only a finite number of write actions can be performed in a finite amount of time such that there always is a last write in any finite period I.

$$set(d) \textbf{ in } I \rightarrow \exists t \in I : set(d) \textbf{ at } t \wedge (\neg set(d)) \textbf{ during } \{t_0 \in I | t_0 > t\} \qquad \text{(FINITEAX)}$$

5.1.1 Examples Specifications

First a simple example:

$$\begin{aligned} & \langle\!\langle x = 4 \wedge em = 5 \wedge read(d_1, 2) \textbf{ at } 3 \rangle\!\rangle \\ & \quad out(d_2, x + 2) \\ & \langle\!\langle x = 4 \wedge em > 5 \wedge read(d_1, 2) \textbf{ at } 3 \wedge set(d_2, 6) \textbf{ at } em \rangle\!\rangle. \end{aligned}$$

We can generalize specifications by using logical variables. For instance,

$$\begin{aligned} & \langle\!\langle x = v \wedge em = t \wedge read(d_1, 2) \textbf{ at } 3 \rangle\!\rangle \\ & \quad out(d_2, x + 2) \\ & \langle\!\langle x = v \wedge em > t \wedge read(d_1, 2) \textbf{ at } 3 \wedge set(d_2, v + 2) \textbf{ at } em \rangle\!\rangle. \end{aligned}$$

The following program never terminates, but sets d infinitely often.

$$\langle\!\langle em = 0 \rangle\!\rangle \textbf{ while } true \textbf{ do } out(d, 0) \textbf{ od } \langle\!\langle \neg term \wedge \forall t_1\, \exists t > t_1 : set(d) \textbf{ at } t \rangle\!\rangle.$$

Termination might also depend on a value read.

$$\begin{aligned} & \langle\!\langle em = 0 \wedge x = 1 \rangle\!\rangle \\ & \quad \textbf{while not}(x = 0) \textbf{ do } in(d_1, x)\, ;\, out(d_2, x + 1) \textbf{ od} \\ & \langle\!\langle\ \forall v, t_1 : read(d_1, v) \textbf{ at } t_1 \rightarrow \exists t_2 \geq t_1 : set(d_2, v + 1) \textbf{ at } t_2 \wedge \\ & \quad ((\exists t : read(d_1, 0) \textbf{ at } t \wedge term) \vee (\forall t : \neg read(d_1, 0) \textbf{ at } t \wedge \neg term)) \rangle\!\rangle. \end{aligned}$$

5.2 Proof system

We briefly present the rules and axioms for our programming language and refer to [14] for more explanation. First in section 5.2.1 we axiomatize the programming language by giving rules and axioms for the primitive statements and the compound programming constructs. Next we give in section 5.2.2 rules and axioms that are generally applicable to any statement. We assume that all logical variables which are introduced in the rules are fresh.

5.2.1 Axiomatization of Programming Constructs

Let $A[exp/x]$ denote the assertion obtained by substituting expression exp for all free occurrences of variable x.

Rule 5.1 (Skip) $$\frac{A[t_0/em] \wedge term \wedge em \geq t_0 \rightarrow C}{\langle\!\langle A \wedge term \rangle\!\rangle \textbf{ skip } \langle\!\langle C \rangle\!\rangle}$$

As mentioned above we assume that t_0 does not occur free in C. Note that execution of the **skip** statement may take 0 time (it is possible that $em = t_0$). An equivalent formulation is:

$$\langle\!\langle \forall t \geq em : C[t/em] \wedge term \rangle\!\rangle \textbf{ skip } \langle\!\langle C \rangle\!\rangle$$

Similarly, rules for assignment and access to device registers are formulated.

Rule 5.2 (Assignment) $$\frac{A[t_0/em, v/x] \wedge term \wedge x = e[v/x] \wedge em \geq t_0 \rightarrow C}{\langle\!\langle A \wedge term \rangle\!\rangle\ x := e\ \langle\!\langle C \rangle\!\rangle}$$

An equivalent formulation for this rule is:

$$\langle\!\langle \forall t \geq em : C[e/x, t/em] \wedge term \rangle\!\rangle\ x := e\ \langle\!\langle C \rangle\!\rangle$$

Rule 5.3 (Read)

$$\frac{A[t_0/em, v/x] \wedge term \wedge read(d,x) \textbf{ at } em \wedge (\neg read(d)) \textbf{ during } (t_0, em) \wedge em > t_0 \rightarrow C}{\langle\!\langle A \wedge term \rangle\!\rangle\ in(d,x)\ \langle\!\langle C \rangle\!\rangle}$$

Remember that x is a local variable (syntactic constraint).

Note that in rules above and below, $em > t_0$ has been used instead of $em \geq t_0$ to express that these statements take some time.

Rule 5.4 (Set)

$$\frac{A[t_0/em] \wedge term \wedge set(d,e) \textbf{ at } em \wedge (\neg set(d)) \textbf{ during } (t_0, em) \wedge em > t_0 \rightarrow C}{\langle\!\langle A \wedge term \rangle\!\rangle\ out(d,e)\ \langle\!\langle C \rangle\!\rangle}$$

which holds because e does not contain global variables (syntactic constraint).

Next follow some rules for the stamemets that influence flow-of-control.

Rule 5.5 (If-then-else) $$\frac{\langle\!\langle A \wedge b \rangle\!\rangle\ S_1\ \langle\!\langle C \rangle\!\rangle,\quad \langle\!\langle A \wedge \neg b \rangle\!\rangle\ S_2\ \langle\!\langle C \rangle\!\rangle}{\langle\!\langle A \rangle\!\rangle \textbf{ if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi } \langle\!\langle C \rangle\!\rangle}$$

Rule 5.6 (If-then) $$\frac{\langle\!\langle A \wedge b \rangle\!\rangle\ S\ \langle\!\langle C \rangle\!\rangle,\quad A \wedge \neg b \rightarrow C}{\langle\!\langle A \rangle\!\rangle \textbf{ if } b \textbf{ then } S \textbf{ fi } \langle\!\langle C \rangle\!\rangle}$$

Rule 5.7 (While)

$$\frac{\begin{array}{l}\langle\!\langle I \wedge term \wedge b \rangle\!\rangle\ S\ \langle\!\langle I \rangle\!\rangle, \\ \forall t_1\, \exists t_2 > t_1 : I[t_2/em] \rightarrow I_1,\quad loc(I_1) = \emptyset,\quad term \text{ does not occur in } I_1\end{array}}{\langle\!\langle I \wedge term \rangle\!\rangle \textbf{ while } b \textbf{ do } S \textbf{ od } \langle\!\langle (I \wedge term \wedge \neg b) \vee (I \wedge \neg term) \vee (I_1 \wedge \neg term) \rangle\!\rangle}$$

The rule for the while statement with a true boolean guard immediately follows from this.

Rule 5.8 (While true)

$$\frac{\langle\!\langle I \wedge term \rangle\!\rangle\ S\ \langle\!\langle I \rangle\!\rangle,\quad \forall t_1\, \exists t_2 > t_1 : I[t_2/em] \rightarrow I_1,\ \ loc(I_1) = \emptyset,\ \ term \text{ does not occur in } I_1}{\langle\!\langle I \wedge term \rangle\!\rangle\ \textbf{while}\ true\ \textbf{do}\ S\ \textbf{od}\ \langle\!\langle (I \wedge \neg term) \vee (I_1 \wedge \neg term) \rangle\!\rangle}$$

Note that there are two cases here: either I includes termination of S and the commitment reduces to $I_1 \wedge \neg term$ or I does not state termination of S and the commitment reduces to $\neg term \wedge (I \vee I_1)$. Finally we consider sequential composition.

Rule 5.9 (Sequential Composition) $\dfrac{\langle\!\langle A \rangle\!\rangle\ S_1\ \langle\!\langle B \rangle\!\rangle,\ \ \langle\!\langle B \rangle\!\rangle\ S_2\ \langle\!\langle C \rangle\!\rangle}{\langle\!\langle A \rangle\!\rangle\ S_1; S_2\ \langle\!\langle C \rangle\!\rangle}$

Next a few rules to reason about timing annotations. Since a timing annotations is not a statement (recall that the statement $[\,TA\,]$ is an abbreviation of **skip**$[\,TA\,]$), we introduce some auxiliary notation. For a timing annotation we use $\langle\!\langle\!\langle A \rangle\!\rangle\!\rangle\ [\,TA\,]\ \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle$.

Axiom 5.1 (Timing Assignment) $\langle\!\langle\!\langle C[me/m] \rangle\!\rangle\!\rangle\ [\,m := me\,]\ \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle$

Axiom 5.2 (Time Measurement) $\langle\!\langle\!\langle C[em/m] \rangle\!\rangle\!\rangle\ [\,?m\,]\ \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle$

Axiom 5.3 (Timing Requirement) $\langle\!\langle\!\langle A \rangle\!\rangle\!\rangle\ [\,R\, me\,]\ \langle\!\langle\!\langle A \wedge em\ R\ me \rangle\!\rangle\!\rangle$

Rule 5.10 (Timing Annotation Sequential Composition)

$$\frac{\langle\!\langle\!\langle A \rangle\!\rangle\!\rangle\ [\,TA_1\,]\ \langle\!\langle\!\langle B \rangle\!\rangle\!\rangle, \langle\!\langle\!\langle B \rangle\!\rangle\!\rangle\ [\,TA_2\,]\ \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle}{\langle\!\langle\!\langle A \rangle\!\rangle\!\rangle\ [\,TA_1; TA_2\,] \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle}$$

Rule 5.11 (Timing Annotation Introduction) $\dfrac{\langle\!\langle A \rangle\!\rangle\ S\ \langle\!\langle B \rangle\!\rangle,\ \ \langle\!\langle\!\langle B \rangle\!\rangle\!\rangle\ [\,TA\,]\ \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle}{\langle\!\langle A \rangle\!\rangle\ S[\,TA\,]\ \langle\!\langle C \rangle\!\rangle}$

With these rules we can derive, for instance, formulae of the form

$\langle\!\langle term \rangle\!\rangle\ S_1[\,?m_2\,]\,;\ S_2[\,?m_1\,]\,;$ **if** b **then** $S_3[\ \leq m_2 + \delta_1\,]$ **else** $S_4[\ \leq m_2 + \delta_2\,]$ **fi**
$\langle\!\langle m_2 \leq m_1 \leq m_2 + max(\delta_1, \delta_2) \rangle\!\rangle$

5.2.2 General Rules and Axioms

The proof system contains several general rules. We only list the rules pertaining to the particulars of our real-time language, and refer to [14] for other rules. The first axiom expresses that a terminating program takes only a finite amount of time.

Axiom 5.4 (Finite Time) $\langle\!\langle em = t_0 \wedge term \rangle\!\rangle\ S\ \langle\!\langle term \rightarrow \exists \delta \geq 0 : em \leq t_0 + \delta \rangle\!\rangle$

For a statement S, let $write(S)$ be the set of shared variables that might be set by S, i.e. the set of variables d such that $out(d, e)$ occurs in S. Similarly, $read(S)$ is the set of shared variables that might be read by S, i.e. the set of variables d such that $in(d, x)$ occurs in S.

Axiom 5.5 (Read Invariance) $\langle\!\langle em = t_0 \rangle\!\rangle\ S\ \langle\!\langle (\neg read(d))\ \textbf{during}\ (t_0, em] \rangle\!\rangle$

provided $d \notin read(S)$.

Axiom 5.6 (Write Invariance) $\langle\!\langle em = t_0 \rangle\!\rangle\ S\ \langle\!\langle (\neg set(d))\ \textbf{during}\ (t_0, em] \rangle\!\rangle$

provided $d \notin write(S)$.

5.3 Verification of a simple control system

We will specify, implement and verify the control program of the following system. A water vessel has limited but unknown influx of water. A pump with a larger capacity than this influx is installed to be able to remove water from it. Activation or deactivation of a pump must prevent that the vessel overflows or runs dry. A device measuring the water level is installed. Critically low and high water levels are selected. The response time d for the control system to switch the pump on or off, tolerable for this system is related to the maximal flow in and out of the vessel, the critical levels and the response deadline.

As a starting point we take the following control requirements:

$$\forall t_1, t_2 : (\text{water critically high})\ \textbf{during}\ [t1, t2] \rightarrow (\text{pump is on})\ \textbf{during}\ [t1 + d, t2]$$

$$\forall t_1, t_2 : (\text{water critically low})\ \textbf{during}\ [t1, t2] \rightarrow (\text{pump is off})\ \textbf{during}\ [t1 + d, t2]$$

For simplicity we assume that two device variables, water and pump, represent the water level and the pump state instantaniously. Assume pump ranges over $\{on, off\}$. The aim is to design a control system satisfying

$$\forall t_1, t_2 : (\mathsf{water} > \mathsf{high})\ \textbf{during}\ [t_1, t_2] \rightarrow (\mathsf{pump} = on)\ \textbf{during}\ [t_1 + d, t_2]$$

$$\forall t_1, t_2 : (\mathsf{water} < \mathsf{low})\ \textbf{during}\ [t_1, t_2] \rightarrow (\mathsf{pump} = off)\ \textbf{during}\ [t_1 + d, t_2]$$

First note that the statements are trivially true if $t_1 + d > t_2$. Hence the above specification is equivalent to

$$CTL_1 \equiv \forall t_1, t_2 : t_1 + d \leq t_2 \wedge (\mathsf{water} > \mathsf{high})\ \textbf{during}\ [t_1, t_2] \rightarrow (\mathsf{pump} = on)\ \textbf{during}\ [t_1 + d, t_2]$$

$$CTL_2 \equiv \forall t_1, t_2 : t_1 + d \leq t_2 \wedge (\mathsf{water} < \mathsf{low})\ \textbf{during}\ [t_1, t_2] \rightarrow (\mathsf{pump} = off)\ \textbf{during}\ [t_1 + d, t_2]$$

That is, the aim is to design a control system Contr such that

$$\langle\!\langle em = 0 \rangle\!\rangle\ \mathsf{Contr}\ \langle\!\langle CTL \rangle\!\rangle,$$

where $CTL \equiv CTL_1 \wedge CTL_2$. The program must keep the pump in the correct state. It can do so by setting the variable pump. Assuming $\forall t : \mathsf{pump}(t) \in \{on, off\}$, axiom (SETAX) leads to the following properties.

$$CA_1 \equiv \forall t : (\exists t_0 \leq t : (\mathsf{pump} = on)\ \textbf{at}\ t_0 \wedge (\neg set(\mathsf{pump}, off))\ \textbf{during}\ (t_0, t]) \rightarrow (\mathsf{pump} = on)\ \textbf{at}\ t$$

$$CA_2 \equiv \forall t : (\exists t_0 \leq t : (\mathsf{pump} = off)\ \textbf{at}\ t_0 \wedge (\neg set(\mathsf{pump}, on))\ \textbf{during}\ (t_0, t]) \rightarrow (\mathsf{pump} = off)\ \textbf{at}\ t$$

Therefore it is sufficient to set the pump at selected moments. Let $CA \equiv CA_1 \wedge CA_2$. Then the top-level specification can be replaced by

$$CP_1 \equiv \forall t_1, t_2 : t_1 + d \leq t_2 \wedge (\mathsf{water} > \mathsf{high})\ \textbf{during}\ [t_1, t_2] \rightarrow \\ \exists t_3 \leq t_1 + d : (\mathsf{pump} = on)\ \textbf{at}\ t_3 \wedge (\neg set(\mathsf{pump}, off))\ \textbf{during}\ (t_3, t_2]$$

$$CP_2 \equiv \forall t_1, t_2 : t_1 + d \leq t_2 \wedge (\mathsf{water} < \mathsf{low})\ \textbf{during}\ [t_1, t_2] \rightarrow \\ \exists t_3 \leq t_1 + d : (\mathsf{pump} = on)\ \textbf{at}\ t_3 \wedge (\neg set(\mathsf{pump}, off))\ \textbf{during}\ (t_3, t_2]$$

Let $CP \equiv CP_1 \wedge CP_2$. Then $CA \wedge CP \rightarrow CTL$, since $(\mathsf{pump} = on)$ **at** $t_3 \wedge (\neg set(\mathsf{pump}, off))$ **during** $(t_3, t_2]$ implies by CA_1 that $(\mathsf{pump} = on)$ **during** $[t_3, t_2]$, and with $t_3 \leq t_1 + d$ this leads to $(\mathsf{pump} = on)$ **during** $[t_1 + d, t_2]$. Hence it remains to implement Contr such that

$$\langle\!\langle em = 0 \rangle\!\rangle\ \mathsf{Contr}\ \langle\!\langle CP \rangle\!\rangle.$$

Next reading the value of water is made explicit. By axiom (READAX) we have

$$CS \equiv \forall t : read(\mathsf{water}, v)\ \textbf{at}\ t \rightarrow \mathsf{water}(t) = v$$

Now define

$$readperiod(t_1, t_2, v) \equiv (t_1 < t_2) \wedge read(\mathsf{water}, v)\ \textbf{at}\ t_2 \wedge (\neg read(water))\ \textbf{during}\ (t_1, t_2)$$

with abbreviations

$$readperiod(t_1, t_2) \equiv \exists v : readperiod(t_1, t_2, v)$$

and

$$readperiod(t_1, t_2, \mathsf{high}) \equiv \exists v : v > \mathsf{high} \wedge readperiod(t_1, t_2, v)$$

$$readperiod(t_1, t_2, \mathsf{low}) \equiv \exists v : v < \mathsf{low} \wedge readperiod(t_1, t_2, v)$$

The control program is now specified by

$CC_1 \equiv \forall t\, \exists t_1, t_2 : t_1 < t \leq t_2 < t + d \wedge readperiod(t_1, t_2)$

$CC_2 \equiv \forall t_1, t_2 : readperiod(t_1, t_2, \mathsf{high}) \rightarrow (\mathsf{pump} = on)\ \mathbf{in}\ [t_2, t_1 + d]$

$CC_3 \equiv \forall t : set(\mathsf{pump}, off)\ \mathbf{at}\ t \rightarrow \exists t_1, t_2 : t \in [t_2, t_1 + d] \wedge readperiod(t_1, t_2, \mathsf{low}) \wedge$
$(\forall t_3, t_4 : t_4 \in (t_2, t] \wedge readperiod(t_3, t_4, \mathsf{high}) \rightarrow set(\mathsf{pump}, on)\ \mathbf{in}\ (t, t_3 + d])$

In addition CC_4 and CC_5 are required, which describe the symmetric case, with high and low, and also *on* and *off*, interchanged.

Let $CC \equiv CC_1 \wedge CC_2 \wedge CC_3 \wedge CC_4 \wedge CC_5$. Then we have the following lemma.

Lemma 5.1 $\mathsf{low} \leq \mathsf{high} \wedge CS \wedge CC \rightarrow CP$.

Proof of lemma:
By symmetry it is sufficient to show CP_1. Lets assume that the premiss of the implication in CP_1 is satisfied and there exists a t_1 and t_2 with

$$t_1 + d \leq t_2 \wedge (\mathsf{water} > \mathsf{high})\ \mathbf{during}\ [t_1, t_2] \tag{1}$$

By CC_1 we know that the water level was read in $[t_1, t_1 + d]$ and obtain t_{11}, t_{21} and v with

$$t_{11} < t_1 \leq t_{21} \leq t_1 + d \wedge readperiod(t_{11}, t_{21}, v) \tag{2}$$

The read action implied by *readperiod* at $t_{21} \in [t_1, t_1 + d]$ falls within the period that the water is assumed high in (1), and thus using (READAX) we obtain $v > \mathsf{high}$. Thus we have

$$readperiod(t_{11}, t_{21}, \mathsf{high}) \tag{3}$$

By CC_2 we obtain $\mathsf{pump} = on$ **in** $[t_{21}, t_{11} + d]$ and we can obtain a t_3 with

$$t_1 \leq t_{21} \leq t_3 \leq t_{11} + d : \mathsf{pump} = on\ \mathbf{at}\ t_3. \tag{4}$$

This implies CP_1 if $(\neg set(\mathsf{pump}, off))$ **during** $(t_3, t_2]$.

Next suppose that $(set(\mathsf{pump}, off))$ **in** $(t_3, t_2]$. We show that also in this case CP_1 holds. Consider the last $set(\mathsf{pump}, off)$ action in the interval $(t_3, t_2]$, which exists according to (FINITEAX). Thus there exists a t_4 with:

$$set(\mathsf{pump}, off)\ \mathbf{at}\ t_4 \wedge t_4 \in (t_3, t_2] \wedge (\neg set(\mathsf{pump}, off))\ \mathbf{during}\ (t_3, t_2]. \tag{5}$$

The $set(\mathsf{pump}, off)$ action in (5) matches the premiss of CC_3 and we conclude that a t_{12} and t_{22} exist with

$$t_4 \in [t_{22}, t_{12} + d] \wedge readperiod(t_{12}, t_{22}, \mathsf{low}). \tag{6}$$

Equations (5) and (6) state that both $t_4 \in [t_{22}, t_{12} + d]$ and $t_4 \in (t_3, t_2]$. We conclude $t_{22} \leq t_2$. However, since a low value was read at t_{22}, (1) and (READAX) imply

$$t_{22} < t_1. \tag{7}$$

Consider the following events: read low water at t_{22}; beginning of period where water is known to be high at t_1; read high water at t_{21}; set pump on at t_3; and set pump off at t_4. Their precedence relations can be inferred from (7), (2), (4) and (5), respectively:

$$t_{22} < t_1 \leq t_{21} \leq t_3 < t_4. \tag{8}$$

In addition to (6) the conclusion of CC_3 expresses a consequence that holds for all $readperiod(m_1, m_2, \mathsf{high})$ with $m_2 \in [t_{22}, t_4]$. It states that if a high water level is read after the read action at t_{22}, the corresponding switching on of the pump must take place after the action on the pump at t_4. From (3) and (8) we know that there is such a *readperiod* with $m_1 = t_{11}$ and $m_2 = t_{21}$. Therefore CC_3 implies

$$set(\mathsf{pump}, on)\ \mathbf{in}\ (t_4, t_{11} + d]. \tag{9}$$

Using (5) and $t_{11} < t_1$ (see (1)) we again obtain the conclusion of CP_1, namely

$$\exists t : set(\mathsf{pump}, on)\ \mathbf{at}\ t \wedge t < t_1 + d \wedge (\neg set(\mathsf{pump}, off))\ \mathbf{during}\ [t, t_2]. \tag{10}$$

End Proof

Hence the control program, Contr, should satisfy the following specification

$\langle\!\langle em = 0 \rangle\!\rangle$ Contr $\langle\!\langle CC \rangle\!\rangle$.

We can consider two simplifications in the specification which correspond to design decisions for the example program. First, the condition $\mathsf{pump} = on$ in CC_2 can be trivially satisfied through the action $set(\mathsf{pump}, on)$. Thus we obtain a stronger version of CC_2

$$CC_2' \equiv \forall t_1, t_2 : readperiod(t_1, t_2, \mathsf{high}) \rightarrow set(\mathsf{pump}, on)\ \mathbf{in}\ [t_2, t_1 + d]$$

We can also introduce a stronger version of CC_3 expressing the additional requirement that a *read* may not take place between a $set(\mathsf{pump}, off)$ action and the last $read(\mathsf{water}, \mathsf{low})$. It will become clear from the example program that this implies non-interleaving read and set sequences.

$$CC_3' \equiv \forall t : set(\mathsf{pump}, off)\ \mathbf{at}\ t \rightarrow \exists t_1, t_2 : t \in [t_2, t_1 + d] \wedge readperiod(t_1, t_2, \mathsf{low}) \wedge \\ (\neg read(\mathsf{water}))\ \mathbf{during}\ (t_2, t]$$

We will continue for now with this stronger specification. We provided the proof of the original, weaker, one because it is more general and similar to what is used for the mine-pump example that will be discussed in the next sections.

We propose the following program as one that satisfies the commitments CC_1, CC_2', CC_3', CC_4' and CC_5' :

```
Pwater_level  =
\\rtvar   =  {t1, t2}
\\var     =  {x}
\\dev     =  {water, pump}
\\const   =  {on, off, high, low, true, d}

        [t1:=0];
        while true do
           [t2:=t1];
           in(water, x) [<t2+d; ?t1];
           if high < x then
               out(pump, on) [<t2+d]
           else
               if x < low then
                   out(pump,off) [<t2+d]
               fi
           fi
        od
```

The program should be read as follows. Initially $t1$ is set to 0 (line 1). On the first sweep of the repetition $t2$ obtains this value in the first annotated statement (line 3). The first time the *in* statement (line 4) is executed it is done so before time d, and $t1$ records the actual execution moment. If subsequently a pump action is required, i.e. either the condition tested in line 5 or line 8 evaluates to true, the state change corresponding to the relevant *out* statement (line 6 or 9 respectively) is realized before the deadline d. In the next sweep t_2 assumes the old value of t_1 (line 3) and t_1 records the execution moment of the next *in* statement that inspects the environment. A possible response to the obtained value must now take place before $t_2 + d$, i.e. within time d from the one but last inspection. And so on for all the following sweeps.

Considering the specification this program can be motivated as follows. CC_1 is satisfied by allowing at most a time d to pass between two consecutive sensor readings. This is expressed in the annotation of line 4. The maximum *readperiod* that ends at a particular reading extends backward to the moment of the previous reading. Thus, CC_2' is realized by imposing a deadline d on the $out(\mathsf{pump}, on)$ statement counted from the execution moment of the one but last water-level inspection. Because with every $read(\mathsf{water}, \mathsf{high})$ there corresponds a $set(\mathsf{pump}, on)$ and vice versa, CC_5' is also satisfied. A formal prove that this program satisfies the specification is given in appendix A.

The mine-pump problem that is introduced in the next section involves control requirements that can be expressed similarly to the requirements that were taken as starting point of this small example. We will use the insights that were obtained in this section also for constructing the solution to this larger problem but we will omit formal proves from now on.

6 Informal specification minepump

We apply our full approach to the mine pump system as described by [6]. In this section we describe this system and give an accurate but informal specification.

The mine-pump system is intended for regulating the ground-water level in a mine through the programmed control of a water pump. This pump must not be operating if the methane-gas level in the mine is too high because of the risk of explosions. Also other gases in the mine are monitored by the system and an operator is provided with information on the conditions in the mine and is given limited control over the pump. In this paper we will not consider the complete system as described in [6] but only the part that is concerned with the pump control. In particular we will omit the interaction with the operator and the monitoring of gases other than methane since they have no impact on the pump operation.

In general, problems like the mine pump involve four parts, namely (1) the environment (i.e. the mine), (2) the devices (sometimes called ironware; an example is the pump), (3) the program-execution platform (or hardware), and (4) the software. The problem is assumed to originate from the desire to control the environment. The part of the specification that expresses the properties of the environment that the control system must realize is referred to as the *top-level requirement specification.* Choices with regard to the implementation of the control solution are made during the subsequent design phase. Initially the solution must be designed as a whole: devices, execution platform and software, because dependencies between these parts exist they cannot be considered as entirely separate problems. For example, the nature of the selected devices will influence the functional as well as the timing requirements on the software and through this the available processing power may interfere with the selection of devices.

However, we will consider the present design and implementation problem to pertain only to the configuration of the hardware and software and we assume that the particulars of the devices are already fixed. In that spirit we will refer to the combination of hardware and software as *the system* or *the control system.* The control system is obtained in two phases: (1) the construction of the program and (2) the selection of the platform configuration and system generation.

The top level specification and the device choices lead to the system requirements.

1. *Top-level specification:* A system must be installed to ensure that the water level in the mine is not getting too high. If there are conditions under which the control system cannot

operate safely the water level is allowed to rise indefinitely but other measures must be taken that are outside the scope of the system.

2. *Device implementation choice:* It is known that there always is a bounded non-negative influx of water. Thus, the amount of water remains the same or increases if no countermeasures are taken. Therefore, the top-level specification is satisfied using a pump with a limited but large enough capacity to outperform influx of water. When the pump is on, the amount of water is decreasing. The pump should not run dry, i.e. the water level should not decrease below a certain level, or the pump should be off.

3. *Device implementation choice:* To detect the water level two sensors are present: a high sensor at a point that is called A for the moment, and a low sensor at a point B. The high and low water sensors will signal to the control system when the water level reaches a sensor: the high-water sensor on the transition from dry to wet and the low-water sensor on the transition from wet to dry.

4. *Requirement on the pump control:* To prevent the water from exceeding certain high and low levels, some timing requirements on the response time of the system must be satisfied. If the water-level is on or above the high point A in the entire time interval $[t1, t2]$ then the pump should be on in $[t1 + dh, t2]$. Similarly, if the water-level is on or below the low point B in the interval $[t1, t2]$ then the pump should be off in $[t1 + dl, t2]$. If the water is between A and B the pump may be either on or off. Because the pump has finite capacity and the influx of water is limited, there is a minimal amount of time, dlh, required for the water to increase from B to A or to decrease from A to B. It may be assumed that $dlh >> hl$ and $dlh >> dh$.

   (Comment: we will take into account that high signals may be given without low signals in between, because the pump may be switched off at any time. Values suggested by [6]: $dl = dh = 10sec$, $dlh = 100sec$.)

5. *Consequence of device implementation choice and safety requirement:* In the mine methane may be present. If the local methane concentration at the pump exceeds a critical level then the pump should be off, otherwise the risk of an explosion becomes unacceptably high. Satisfaction of his requirement has priority over the pump-control requirements.

6. *Device implementation choice:* A methane sensor is used to measure the concentration of methane in the mine atmosphere near to the pump. The methane level increases or decreases with a maximum rate. Therefore some response time remains when the methane level is still sub-critical by a certain amount. This response time is called dc. The device must be polled by the control system.

   (Comment: values suggested by [6]: $dc = 30msec$.)

7. *Requirement on the pump control:* If the methane level is above the previously mentioned sub-critical value in $[t1, t2]$ then the pump should be off in $[t1 + dc, t2]$ no matter what the water level is. If the methane level falls under this sub-critical level, then the pump can be switched on again. Thus, the condition under which the pump must be *on* changes into: if water on or above high and methane under the sub-critical value in the entire interval $[t1, t2]$ then the pump must be on in $[t1 + dh, t2]$.

8. *Consequence of unreliable pump-device:* Measures should be taken to inspect proper functioning of the pump. Having the pump on for longer then df while it is not functioning correctly is dangerous and should at all times be avoided (higher priority than water-level control). Fault-hypothesis: the pump may fail. If it fails it generates a lower water flow than when it is functioning properly and is switched on. The residual flow is under a value min. If the pump is faulty it remains so forever.

9. *Device implementation choice:* A (reliable) water flow sensor is installed to check the correct operation of the pump. This sensor must be polled. On inspection it provides the value for the flow of water. If the pump is on and the water flow is less than min, the pump is faulty.

10. *Requirement on the pump:* If the pump is faulty and on it should be switched off. The pump should remain in off position forever. It must be ensured that: if pump faulty and on in $[t1, t1 + df]$ then pump off in $[t1 + df, inf)$.

    (Comment: value suggested by [6]: $df = 100msec.$)

Since fault tolerance is not an issue in this paper we will assume that the platform has no failures.

7 Interaction with the environment

The way the control system interacts with the environment that it controls, in particular the programmer's view of such interaction, is an important issue that must be addressed before we discuss a program for the mine-pump problem.

Here we distinguish two modes for receiving information from the environment: polling of sensor values and signal (interrupt) based handling of sensor information. The former mode can be included in our programming model through the introduction of special variables (device registers) that can only be inspected (i.e. a read-only variables). We will adopt this view and assume that reading such a variable yields a particular sensor value at the very moment of inspection. This assumption is for simplicity and does not affect the essence of our approach.

The latter mode, interrupt based interaction, is fundamentally more problematic. In practice interrupt-handling mechanisms are employed that may result in loss of interrupts if previous ones are not handled timely. Our experience with the case study presented in this paper is that loss of messages may cause problems with the program semantics if it is not under explicit control of the program. To avoid such problems we have intentionally introduced infinite FIFO message buffers and we assume that the interaction between devices and software processes can be described using regular message passing via special "device channels". Devices can thus be viewed as "physical" processes that are connected to software processes through such channels. This model has the advantage that a device is equivalent to, and can be specified through, a software process that yields the same set of possible communication behaviors as the device.

Infinite buffers obviously can not be implemented, and part of guaranteeing the existence of an implementation is an analysis of the resource requirement of a program and showing that it can be satisfied. Thus, obviously, not only timeliness is a concern during scheduling, but also resource availability and utilization.

A specification of the possible behaviors of the environment that has to be controlled must be input for program construction as well as system generation. The programmer requires a specification of possible environment behavior to be able to construct a solution to the control problem. The scheduler must use the same information to establish feasibility of an implementation of this solution on the provided platform. We mentioned before that the environment may be considered equivalent to a software process and its behavior can be described using our programming language. Since programs do not refer to any execution mechanism they can specify the behavior of the environment in an abstract fashion. The scheduler, naturally, does not schedule the environment but uses this specification as input to obtain a schedule for the control program.

As an illustration of this, consider the high/low water sensors described in the previous section. We can view these sensors as one "physical" process with two communication channels that has the same set of communication behaviors as the following process:

```
Pphysical      =

\\rtvar = {t}
```

```
\\var   = {random}
\\const = {0,true, dlh, delta}
\\chan  = {high_sensor, low_sensor}

        [t:=-dlh];
        while true do
                while random do
                        [>t;?t];    // water surface reaches high sensor
                        send(high_sensor,[!t]);
                        [<t+delta]
                od;
                [t:=t+dlh];
                while random do
                        [>t;?t];    // water surface reaches low sensor
                        send(low_sensor,[!t]);
                        [<t+delta]
                od;
                [t:=t+dlh]
        od
```

The behavior of this process follows from the program semantics. Nothing of its execution speed is known: progress should be considered arbitrary for as far as it is not specified. (As mentioned before we assume, however, liveness.) To express the nondeterministic nature of the environment a boolean variable *random* yields at inspection an arbitrary choice between true and false.

The timing parameter t that is written with a sent message, represents the moment at which the water actually reaches the level communicated in the message. It is important to understand the nature of this timing parameter. This program does not imply that the time value is included in the message at run-time. Timing parameters are part of the annotation which just describe the timing behavior of the program. If the run-time system does not need this information or only a compiled version of it, timing parameters will not actually be sent.

The interpretation of the timing parameter (the fact that it corresponds to the moment a certain water level is reached) must be used by the programmer to construct his control program (just like a programmer needs to interpret the variable x in a function call $sin(x)$ as a particular angle in radians to be able to construct a meaningful program). During system construction this interpretation is not relevant and only the environment specification as it stands must be supplied (e.g. to the scheduler or schedulabilty analyzer).

Thus the above program corresponds to the following behavior of the environment: When the control system is started the water will reach one of the sensors, say the high one, after an unknown period of time. This results in a message on the high-sensor's channel. Subsequently the water-level will pass the sensor an arbitrary number of times in succession, with an unspecified rate. This is due to the fact that the pump may be switched off at arbitrary times: to keep the system robust and more generally applicable we assume very little about the behavior of the environment. A message will be sent at each passage and messages from the sensor may, therefore, be arbitrary in number and rate. If the water surface subsequently reaches the low sensor there is at least an amount of *dlh* time between the last high-sensor crossing and low-sensor crossing, because the water has to decrease in level substantially. Then, the low-sensor can be triggered an arbitrary number of times after which, again, the high sensor will be triggered at least a *dlh* amount of time after the last low-sensor passage. This sequence of events continues indefinitely.

Note that there is some information on the device latency in this program. Whether this should be included and how large it is, depends on where the interface with the control system and the environment is envisaged. Here it is assumed that in addition to a delay in the *high_sensor* and *low_sensor* channels, which are assumed part of the control system, there is a device latency that is maximally delta: the sensors place a message onto the channel within *delta* time, where delta is a given constant. In the following we will make the simplifying assumption that $delta = 0^+$.

The specification leaves room for a large number of possible message sequences with their timing. Naturally, the control program must respond appropriately to all of the allowed behaviors of the environment. In particular it must be shown that a program exists which, among other things, can handle the arbitrarily high number of messages that may occasionally arrive, and it must be shown that a feasible implementation of the program exists on a given platform. The latter is the task of the system generator. It must obtain the environment specification as input in addition to the control program and platform specification to determine whether an implementation of the control program is feasible and to generate this very implementation. Typically the system generator must know the (maximal) delay of the message in the channels (platform specification) and the latency of the device (environment spec), e.g. to compute if the system can satisfy a possible deadline on the response.

8 Design decisions

During the design phase decisions have to be made on the division of the program into parallel processes and the assignment of subtasks to each of them. In the following we describe the rationales that have led to the choices made to arrive at the implementation that is presented in this paper. At this stage we have no systematic method to derive real-time programs from a specification. Development of such a real-time programming method is considered an interesting subject for future research.

The only externally observable effect of the program is that the pump is being switched on and off at appropriate moments. There are three aspects of the environment that have their bearing on the pump: (1) the water level, (2) the methane level, and (3) failures of the pump. These physical processes are made visible to the control software by the sensors. The nature of the environmental conditions determines if and how urgently a response of the system is required. The measurements of the conditions are independent problems both in their function as well as timing. Thus we shall introduce a separate process for monitoring each sensor. These processes determine from the sensor state if an action should be taken on the pump. Thus we have sensor processes *Pmethane*, *Pwater* and *Pflow*. *Pwater* is message driven, i.e. it is triggered by messages passed to the control system by the environment. It is therefore not of the form of the example described in section 5.3. The other involve polling loops that are likely similar to what we described there.

Although the handling of each sensors is an inherently independent activities, the resulting actions on the pump must be coordinated. Because we only have local device registers in our programming language there can be only one process that switches the pump, and it must make decision to that effect on the basis of messages passed by the processes that monitor the sensors. Therefore a process that sets pump, *Ppump*, is introduced.

Ppump must be informed by the sensor processes about the following relevant conditions in the environment:

- the water level may be too high,
- the water level may be too low,
- the methane level is too high,
- the methane level is ok,
- the pump has failed.

Note that the condition "the water level may be too high" is true after the high sensor has been reached and cannot be denied until the low sensor is reached, then it is replaced by "the water level may be too low". If the former condition holds, the pump should be on, if the latter holds, it should be off (within the response time margin).

To enable an implementation of the control condition we introduce in *Ppump* (boolean) program variables that have the same value as the corresponding assertions about the environment at least within the appropriate reaction time. e.g. :

"the water level may be too high" $during\ [t1, t2]$
$\longrightarrow (water\_may\_be\_too\_high\ =\ true)\ during\ [t1 + d, t2]$

On the basis of the value of these variables a decision is taken with regard to the action on the pump. The control specification implies that the following assertion, P, must hold within the deadline of each change in the relevant environmental condition:

$$\begin{aligned} P = \ & ((water\_may\_be\_too\_low\ or\ \neg safe) \longrightarrow pump = off)\ and \\ & ((water\_may\_be\_too\_high\ and\ safe) \longrightarrow pump = on), \end{aligned}$$

where

$$safe = \neg(methane\_too\_high\ or\ pump\_is\_faulty)$$

and

$$\neg(water\_may\_be\_too\_high\ and\ water\_may\_be\_too\_low).$$

Ppump will have to maintain the state of the boolean variables and set the pump variable to satisfy assertion P. Thus *Ppump* will be able to receive messages from all the processes monitoring a sensor and will implement a response. The messages can be annotated with the response time for the condition that is communicated.

There is one additional complication. Although measuring water flow of the pump and establishing failure of the pump is an activity that is in principle independent from the pump-control, the water flow measurements are only necessary when the pump is on and the the measurements must be synchronized with *Ppump* to be sure that the pump state is known at the moment the flow is measured. After all it must be established that if no flow is measured this is not simply due to the switching off of the pump. This problem will be appropriately addressed in the implementation discussed in the next section.

9 Implementation

In this section we will present an implementation of the mine pump control program. The processes are sometimes obtained in several steps. Whenever we write $P = ..$ we describe the final version of a particular process. On the other hand when we write $P \sim ..$ we are discussing an intermediate version.

The following channels and constants will be present (all local entities will be mentioned with the corresponding process).

```
chan  =  {methane, water, check, check_flow, high_sensor, low_sensor}
const =  {0, true, false, high, low, too_high, ok_again, max, min, on, off,
          dh, dl, dlh, dc, df}
```

9.1 Process handling CH4 sensor

The process *Pmethane* determines from a sensor reading if the methane level becomes too high for safe operation of the pump or if it reaches an acceptable level after having been too high. A boolean variable *high_methane* is updated to record the state of the environment. Whenever this boolean changes a message is sent to the pump process. The message contains a timing parameter prescribing the deadline within which the messages must be handled.

```
Pmethane    =

//rtvar = {t1,t2}
//var   = {methane_level, high_methane}
//dev   = {methane_sensor}

        [t1:=0];
        high_methane := false;
        while true do
                [t2:=t1];
                in(methane_sensor, methane_level) [<t2+dc; ?t1];
                if (methane_level<max) and (high_methane=true) then
                     high_methane := false;
                     send(methane, ok_again, [!t2+dh])
                else if not (methane_level<max) and (high_methane=false)
                         high_methane := true;
                         send(methane, too_high, [!t2+dc])
                     fi
                fi
        od
```

Note that this timing annotation pattern is similar to the one described in section 5.3, apart from some (not unimportant) change in detail: (1) only one critical level exists, (2) the actions on the pump are achieved through sending a message, (3) the state is recorded (variable *high_methane*) and only changes in this variable give rise to actions on the pump, (4) there are two different response deadlines related to the level becoming too high and acceptable respectively. The deadline on the pump action is passed as a timing parameter in the message. The message is handled by the receiving process which will continue the sequence of actions and realize a proper response within this specified deadline. Note, that the deadline on the sensor inspection is $t2 + dc$ which is the minimum of $t2 + dc$ and $t2 + dh$ because it must always be possible to verify the methane condition within the shortest response deadline.

9.2 Process handling high and low water level messages

Because the monitoring of the water level is message driven, the process is quite different from the polling loop of the methane process and the example of section 5.3. *Pwater* alternately expects the water to be high and low. Since it is assumed that the pump is off initially the water will be rising. Therefore the first message that may arrive is one idicating high water. Both assertions *the water may be too low* and *the water may be too high* may assumed to be initially false.

As soon as a high water level is detected a message is sent to *Ppump* indicating this condition and a response must occur within the deadline *dh*. In general, only the first low message after a high one is relevant and requires a response and all other low-messages can be discarded. Thus, the buffer that keeps messages from the low sensor is flushed before the water can reach the low-sensor to make sure that only a relevant new message to this effect will cause a response. Such a low-sensor message is handled in a similar fashion as the high one but with a different deadline, namely *dl*.

```
Pwater      =
//rtvar = {t}
//var   = {}

        while true do
                receive(high_sensor, [?t]);
                send(water, high, [!t+dh]);
                flush(low_sensor);
                [<t+dlh];
                receive(low_sensor, [?t]);
                send(water, low, [!t+dl]);
                flush(high_sensor);
                [<t+dlh]
        od
```

It is interesting to note that in the implementation of this program only one-place buffers that retain the oldest message are required for the device channels, even though message rates from the environment can be arbitrarily high.

9.3 Process measuring pump faults

If we disregard the earlier assumed process structure one would, inspired by the example of section 5.3, attempt the following solution.

```
Pcheck      ~
         [t1:=0];
         pump_is_faulty := false;
         while true do
            [t2:=t1];
            in(pump, status) [<t2+df;?t1];
            if status=on then
                 in(flow_sensor,x);
                 pump_is_faulty := (x<min);
                 if pump_is_faulty=true then
                      out(pump,off) [<t2+df]
                 fi
            fi
         od
```

There are two problems with this implementation. First, the pump process is receiving messages from the other processes and must be ready to do so. Thus, it is not possible to integrate the above polling loop completely into this process. Second, we would like to place the pump and flow sensor management in separate processes, to make the process structure reflect the physical distribution of the system and separate the concerns of measuring the flow and controlling the pump. A simple strategy to solve these problems is to cut the above process into three parts that are allocated to *Pcheck*, *Pflow* and *Ppump*, respectively. The relevant steps in the procedure, (1) starting an iteration, (2) determining the pump state, (3) checking the flow and (4) switching the pump off if necessary, are allocated to these processes and are always executed in succession through the passing of messages.

```
Pcheck      =

//rtvar  = {}
//var    = {}

         while true do
               send(check)
         od
```

```
Pflow       =

//rtvar = {t}
//var   = {x}
//dev   = {flow_sensor}

        while true do
                receive(check_flow);
                in(flow_sensor, x);
                send(flow, (x<min))
        od
```

The third part must be integrated into *Ppump*:

```
Ppump       ~
        [t:=0];
        pump_is_faulty := false;
        while true do
                receive(check);
                [t:=t1+df];
                in(pump, status) [<t; ?t1];
                if status=on then
                      send(check_flow);
                      recieve(flow, pump_is_faulty);
                      if pump_is_faulty=true then
                           out(pump, off) [<t]
                      fi
                fi
         od
```

Note, that *Pcheck* does not contain any timing annotation. Still, the processes combined satisfy the timing constraints, e.g. the constraint that the pump is checked within each interval *df*. This implies (through the precedence constraints between the processes) that *Pcheck* sends a message that initiates the checking procedure frequently enough.

9.4 Process handling pump

The pump process handles the messages from the three processes, *Pmethane*, *Pwater* and *Pcheck* and carries out the required pump actions within the specified deadlines. It must icorporate the actions described in the previous subsection. The complete program is:

```
Ppump       =

//rtvar = {t, t1}
//var   = {value, water_may_be_too_high, water_may_be_too_low,
//         methane_is_too_high, pump_is_faulty, status}
//dev   = {pump}

        [t1:=0];
        water_may_be_too_high := false;
        water_may_be_too_low := false;
        methane_is_too_high := false;
        pump_is_faulty := false;
        while true do
               select receive(water, value, [?t])
                       do
                                 if value=high then
```

```
                                        water_may_be_too_high := true;
                                        water_may_be_too_low  := false;
                                        if methane_is_too_high=false then
                                               out(pump, on) [<t]
                                        fi
                                else // value=low
                                        water_may_be_too_low  := true;
                                        water_may_be_too_high := false;
                                        out(pump, off) [<t]
                                fi
                        od
                or      receive(methane, value, [?t])
                        do
                                if value=too_high then
                                        methane_is_too_high := true;
                                        out(pump, off) [<t]
                                else //value=ok_again
                                        methane_is_too_high := false;
                                        if water_may_be_too_high=true then
                                               out(pump, on) [<t]
                                        fi
                                fi
                        od
                or      receive(check)
                        do
                                [t:=t1+df];
                                in(pump,status) [<t; ?t1];
                                if status=on then
                                      send(check_flow);
                                      recieve(flow, pump_is_faulty);
                                      if pump_is_faulty=true then
                                            out(pump, off) [<t]
                                      fi
                                fi
                        od;
        od

//  When a message is handled by this process the following assertion holds:
//
//     P  =      ((water_may_be_too_low or not safe) -> pump=off) and
//               ((water_may_be_too_high and safe)   -> pump=on)  and
//               ((safe = not (methane_too_high or pump_is_faulty )) and
//               not (water_may_be_too_high and water_may_be_too_low)
```

Note that for clarity we have introduced the variable *water_may_be_too_high* as well as the variable *water_may_be_too_low*, while only one is actually required.

The present implementation of the pump process solves the synchronization problem mentioned before: the only process that can change the state of the pump, *Ppump*, is blocked during the flow measurement. Thus, it is known that the pump is on when the flow of the pump is being measured by *Pflow*.

10 System generation

In the following sections we will give examples of implementations of the mine-pump control program on several platforms. By platform we mean the processors and their interconnection as well as the execution mechanism that is employed. In each case we give the conditions under which the program is feasible. We give no formal proof of this feasibility.

We consider two models:

1. Off-line (static) scheduling. We consider two cases, maximum and minimum parallelism. We assume a minimal but sufficient number of (bidirectional) communication lines to allow for direct process to process message passing.
2. On-line preemptive priority scheduling. We will only consider the most interesting case of minimum parallelism for this model.

We will describe each of these models in detail.

First we analyse the programs process and scheduling block structure, and find critical execution paths in the program. On the basis of this analysis we study the two cases: maximum and minimum parallellism. The on-line scheduling approach will also benefit from the analysis done for the off-line case. The schedulability will be studied for both approaches and a numerical example will be given.

11 Off-line scheduling

We assume the following execution model. An off-line calculated schedule is passed with the executable image of each process to an on-line dispatcher. The task of the off-line scheduler is (1) to divide a process in scheduling blocks and (2) to assign to each process a number of time slots for execution. The dispatcher will deal with these slots as follows. At the moment the time slot for a process is reached, execution of this process will start or resume, but only if it is ready to run. If the process is not ready, i.e. an awaited message has not yet arrived, the time slot will be relinquished. When the end of an executing block is reached, execution will be suspended until the next time slot for that process. At the end of a block the control will be returned to the dispatcher either through a blocking statement contained in the program (i.e. *receive* statement) or a preemption instruction that is inserted in the machine code during system generation. Although in general this model requires several refinements that describe how alternatives in block sequences are handled, the present level of detail is sufficient for the problem at hand.

The off-line scheduler cannot predict variations of behavior of the environment that remains within the environment specification. The schedule must therefore be based on worst case assumptions. It must, e.g. , be taken into account that messages from the water-level sensor may arrive at times and rates described by the program in section 7.

We will proceeed by analyzing the program in several steps and infer schedules for the maximum and minimum parallellism cases.

11.1 Program analysis: process structure

In Figure 1 we have depicted the process structure of the minepump program. The processes, communication channels and device variables are drawn. The channels are directed arrows indicating the directions in which the messages are send.

11.2 Program analysis: scheduling blocks

For scheduling purposes the minepump program it is divided into beads. Beads are the smallest program parts that must be analyzed on their execution duration. To reduce the size of the scheduling problem, the off-line scheduler will compute time slots for groups of such beads which are the previously mentioned scheduling blocks. We will not address the problem of recognizing and grouping beads, nor how a schedule may be found. It is possible that a block contains alternative groups of beads, but, naturally, the worst-case behavior must be taken into account.

New bead starts at either the beginning of a process, at a time measurement, at an earliest start-time requirement or the reception of a message. A bead ends at the sending of a message, at

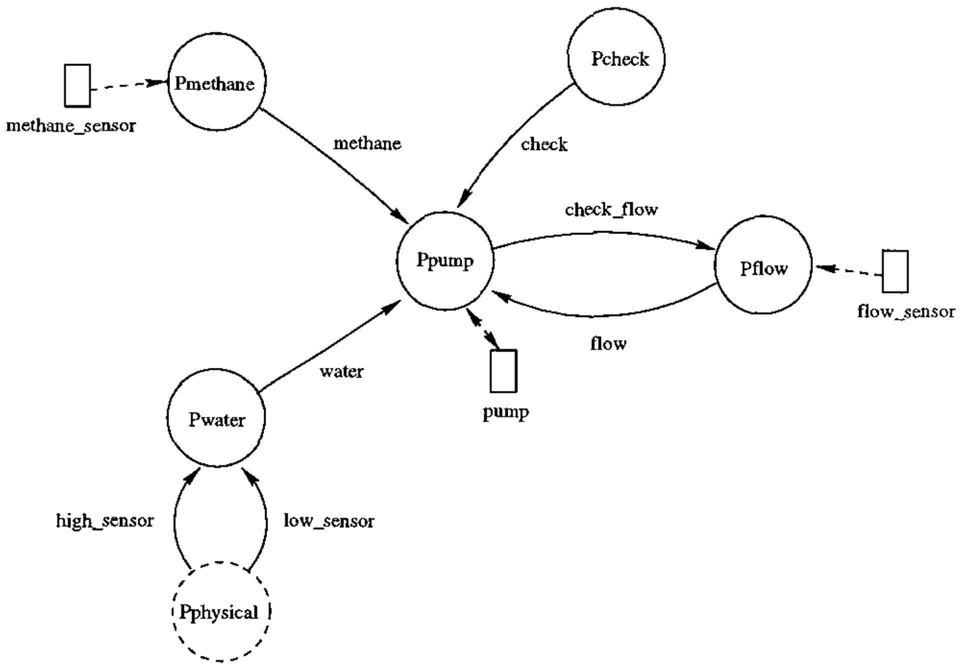

Figure 1: The process graph for the mine pump.

a deadline requirement or the end of a process. When beads are combined into scheduling blocks, earliest start times may be moved to the beginning of the block and deadlines may be moved to the end. Because time measurements may be used to compute both deadlines as well as earliest start times they must be known more precisely. However, it is in practice very difficult to dispatch a bead exactly at the position of the measurement. Therefore we assume that for a bead in a block preceeding such a measurement, colored black in the figure, a best and worst case execution time is known. If these times are not equal constraints involving the measurement may not always be satisfiable. For simplicity we assume here that these times may be considered equal by the scheduler. The scheduler can then straightforwardly infer the value of a time measurement from the placement of a scheduling block. Blocks have a structure as depicted in Figure 2.

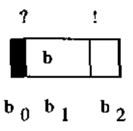

Figure 2: An example of a scheduling block that contains a time measurement at the end of the first bead (indicated by ?) and a constraint on the end of the second (indicated by !).

Blocks in a process are labeled. Beads in a block have the same label with an additional subscript to distinguish them. Time measurements or requirements on a bead-boundary are indicated by question and exclamation marks, respectively. We denote execution durations of beads as follows:

- duration of bead preceeding a measurement: $Tb_0 = duration(b_0)$
- maximum duration of regular bead : $Tb_n = max\_duration(b_n)$
- maximum duration of total block: $Tb = Tb_0 + ... + Tb_n$

Figure 3 shows the bead and block structure of the various processes in the program. All processes consist of an initiallization part and a cyclic part. This cyclic part in some processes is triggered by a message (e.g. *Pflow*), or several messages in sequence (e.g. *Pwater*) or message alternatives (e.g. *Ppump*). Some processes send a message or have the option to send a message during execution of a block. Optional messages or message alternatives are indicted by dotted lines. Two processes, *Pmethane* and *Pcheck*, have a cyclic part that is not triggered by a message. Their rate of execution is arbitrary but such that timing constraints are satisfied.

Code analysis should provide maximum execution durations of the various beads. To find tight upper bounds for such durations for realistic programs and modern processors is not a simple problem. However, some encouraging research has been done in this regard. We will not further address this issue here.

11.3 Program analysis: critical paths

To investigate possible schedules, the timing constraints are considered in more detail. We can discern critical execution paths, i.e. sequences of statements across processors that correspond to worst case execution times towards a deadline. These paths are depicted in Figure 4. In fact there are four such paths: one corrseponding to the response to a critical or subcritical methane level, two corresponding to detection of high and low water, and one corresponding to failure detection of the pump.

In our search for a feasible schedule we will consider situations where the blocks of the cyclic parts of the processes are dispatched strictly periodically, starting from some offset. Note that there are many alternatives to a strictly periodic schedule: the program does not state any preference and a periodic schedule is strictly an implementation choice. It is not required for correctness.

As can be seen from the figure, the periods determine how much time remains for the processes to make their deadlines and thus they are very important in the scheduling analysis. Each critical path in Figure 4 yields a constraint that corresponds to meeting the deadline during normal periodic execution. Another requirement is that the initialization block of each process must be executed within this offset. Thus, constraints can be inferred for meeting the deadlines after initialization. These are not depicted in the figure.

We consider the constraints per process, first the initialization constraint and then the constraint(s) corresponding to the cyclic execution. The latter ones can be directly compared with the figure. For each message a message-passing time is introduced, Mx with x the channel name (no distinction between individual messages on a channel needs to be made here). This passing time is the time between the sending of the message and the moment the receiving block is dispatched.

Pmethane

Start times $i_m + x * r_m$ with $x \in \mathbb{N}$

Constraints

Initial: $i_m + Tm1_0 + Tm1_1 + Mm + Tp2_0 \leq dc$

Cyclic: $r_m + Tm1_1 + Mm + Tp2_0 \leq dc$

Pwater

Start times $i_w + x * r_w$ with $x \in \mathbb{N}$

Constraints

Initial: $i_w + Tw1_0 + Mw + Tp1_0 \leq dh$

$i_w + Tw1_0 + Tw1_1 \leq dlh$

Cyclic: $Mh + Tw1_0 + Mw + Tp1_0 \leq dh$

$Ml + Tw2_0 + Mw + Tp1_0 \leq dl$

$Mh + Tw1_0 + Tw1_1 \leq dlh$

$Ml + Tw2_0 + Tw2_1 \leq dlh$

Pcheck

Start times $i_c + x * r_c$ with $x \in \mathbb{N}$

Constraints

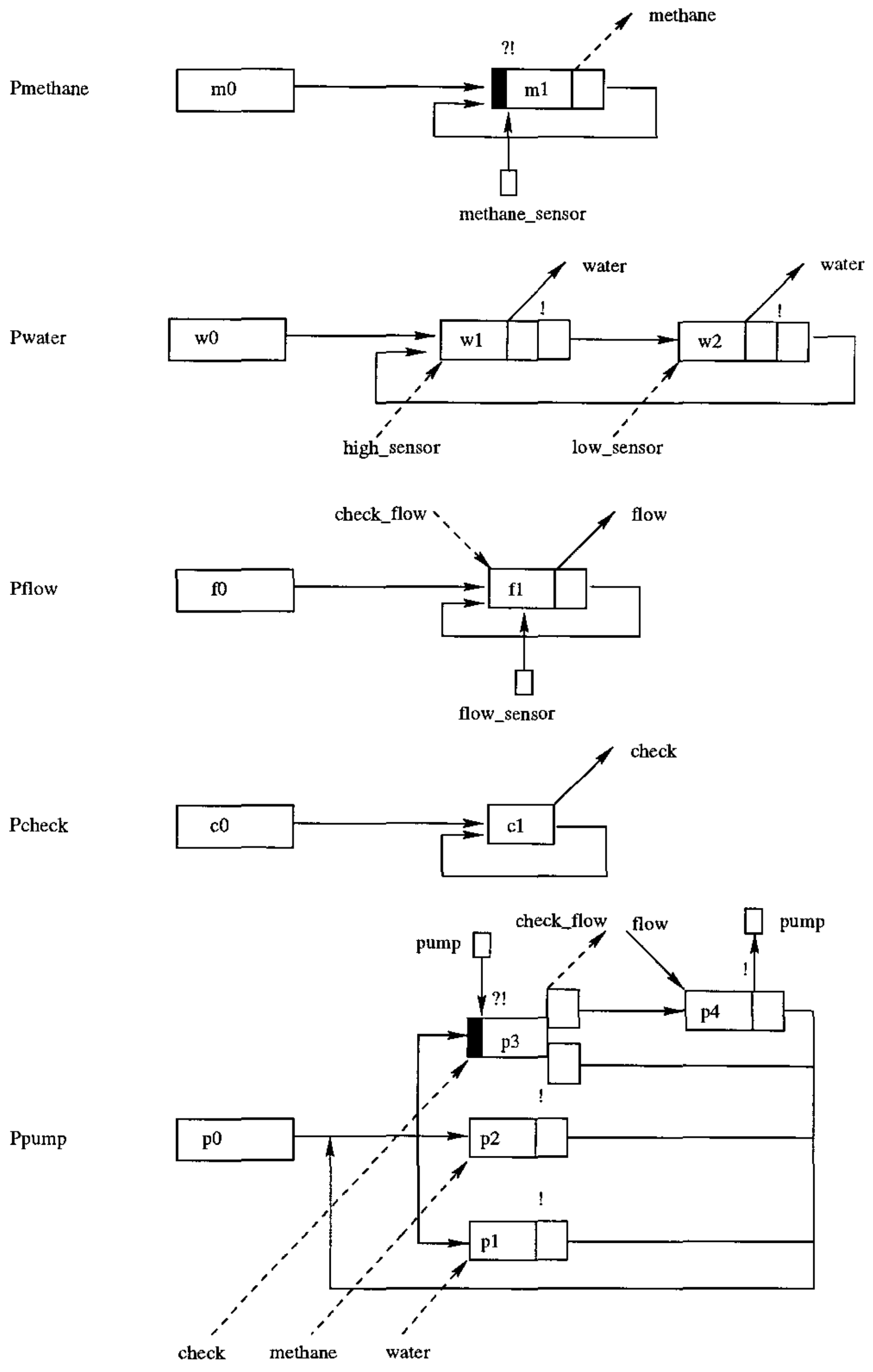

Figure 3: Block structure of the processes

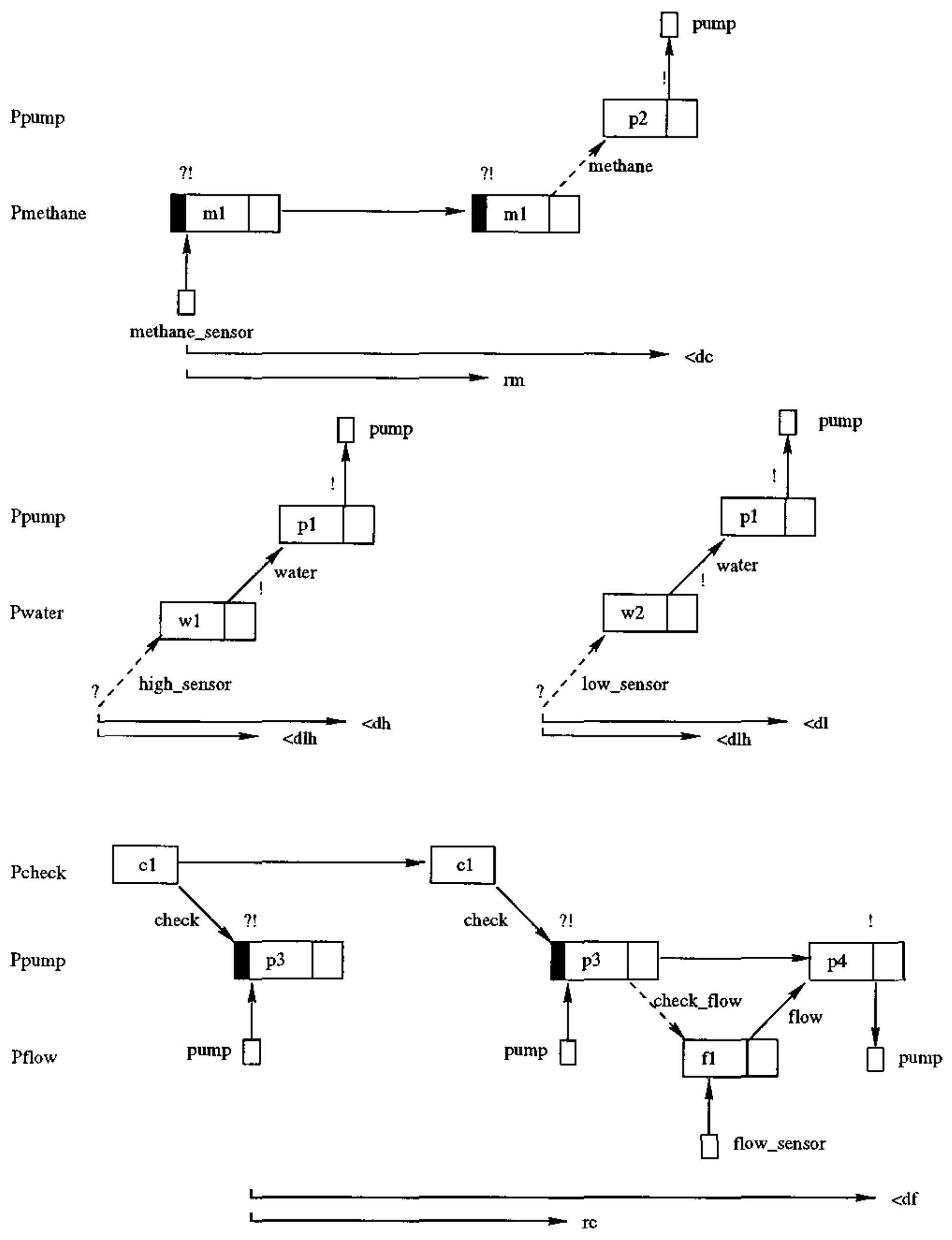

Figure 4: The critical paths through the program.

Initial: $i_c + Tc1 + Mc + Tp3_0 + Tp3_1 + max(Mcf + Tfl_0 + Mf, Tp3_2) + Tp4_0 \leq df$
Cyclic: $r_c + Tp3_1 + max(Mcf + Tfl_0 + Mf, Tp3_2) + Tp4_0 \leq df$

Pflow
Start times $i_f + x * r_c$ with $x \in \mathbb{N}$
Constraints
Initial: $i_f + Tfl_0 + Mf + Tp4_0 \leq df$

Ppump
Start times $i_{p1} + x * r_w$ with $x \in \mathbb{N}$
$i_{p2} + x * r_m$ with $x \in \mathbb{N}$
$i_{p3} + x * r_c$ with $x \in \mathbb{N}$
$i_{p4} + x * r_c$ with $x \in \mathbb{N}$
Constraints
Initial: $i_{p1} + Tpl_0 \leq dh$
$i_{p2} + Tp2_0 \leq dc$
$i_{p3} + Tp3_0 + Tp3_1 + max(Mcf + Tfl_0 + Mf, p3_2) + Tp4_0 \leq df$
$i_{p4} + Tp4_0 \leq df$

Note that the message handling parts of *Pflow* and *Ppump* have periods that correspond to the periods of the processes that send the messages. Their contribution to the cyclic activities is already contained in the expressions for these sending processes. Only initialization constraints remain for these processes.

11.4 Off-line scheduling and maximum parallellism

The hardware configuration for the maximum parallelism case is given in Figure 5. A separate processor is allocated to each device. The communication channels are positioned to allow for the required communication between the processes handling the devices. The allocation of processes to processors follows naturally in this hardware configuration.

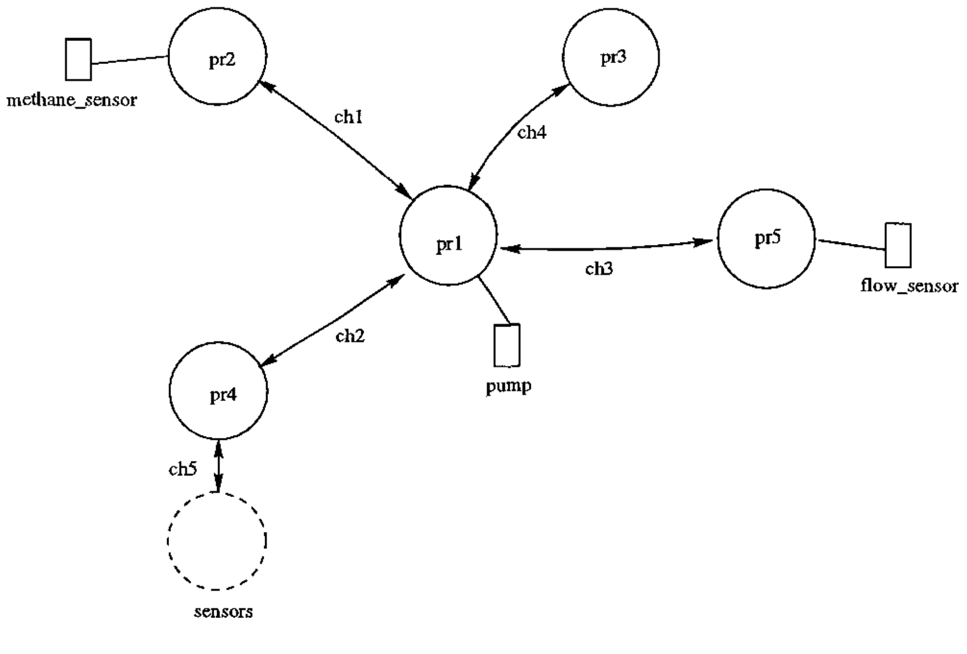

Figure 5: The maximal hardware configuration.

Ppump	$\longrightarrow$	pr1	check_flow	$\longrightarrow$	ch1
Pmethane	$\longrightarrow$	pr2	methane	$\longrightarrow$	ch2
Pcheck	$\longrightarrow$	pr3	flow	$\longrightarrow$	ch3
Pwater	$\longrightarrow$	pr4	water	$\longrightarrow$	ch4
Pflow	$\longrightarrow$	pr5	check	$\longrightarrow$	ch5
			high_sensor	$\longrightarrow$	ch6
			low_sensor	$\longrightarrow$	ch6

In the previous section we have already made the assumption that a schedule for each process is strictly periodic. The message delays used in the formulae presented there, included the time for the receiving blocks to be dispatched. We will search for a schedule where these times are minimal: blocks that handle an incoming message can be dispatched as soon as a message has arrived at the processor. Under this assumption, all message delays Mx become equal to the delay M on the channel (this delay is assumed to be the same for all channels). The only messages for which this is not possible are the high and low water messages which are not under the control of the scheduler. The worst case handling delay must be assumed here: $Mh = Ml = M + r_w$. The condition that the blocks receiving a message must be dispatched immediately leads to the following constraints on the period off-sets:

$$\begin{aligned}
&i_{p1} = i_w + Tw1_0 + M,\\
&i_{p2} = i_m + Tm1_0 + Tm1_1 + M,\\
&i_{p3} = i_c + Tc1 + M,\\
&i_{p4} = i_{p3} + Tp3_0 + Tp3_1 + max(2M + Tfl_0, Tp3_2)\\
&i_f = i_{p3} + Tp3_0 + Tp3_1 + M,
\end{aligned}$$

and we have to show now that choices for the parameter $i_m, r_m, i_w, r_w, i_c, r_c, i_f$ and i_{p1}, i_{p2}, i_{p3} exist such that the scheduling blocks can be accomodated. The only problem in the maximum parallellism case is the proper interleaving of the blocks of the pump process. This requires periods r_m, r_w and r_c that are commensurate and that at worst all three branches of the pump process must be accomodated within a period $lcd(r_m, r_w, r_c)$. In such a period we handle the methane messages first and the water messages last, because the dealines are shortes for the former and longest for the latter. The slots for the respective blocks of *Ppump* are chosen contiguous, yielding the additional conditions:

$$\begin{aligned}
&\exists x, y \in \mathbb{N}\\
&i_{p3} = i_{p2} + x * lcd(r_m, r_c, r_w) + Tp2\\
&i_{p1} = i_{p3} + y * lcd(r_m, r_c, r_w) + Tp3_0 + Tp3_1 + max(2M + Tfl_0, Tp3_2)
\end{aligned}$$

A schedule of this type is depicted in Figure 6.

The only remaining conditions for the decribed schedule to be feasible is that each of the processors can handle the blocks within their periods. This yields the following requirements.

Pmethane	$i_m \geq Tm0$
	$r_m \geq Tm1$
Pwater	$i_w \geq Tw0$
	$r_w \geq max(Tw1, Tw2)$
Pcheck	$i_c \geq Tc0$
	$r_c \geq Tc1$

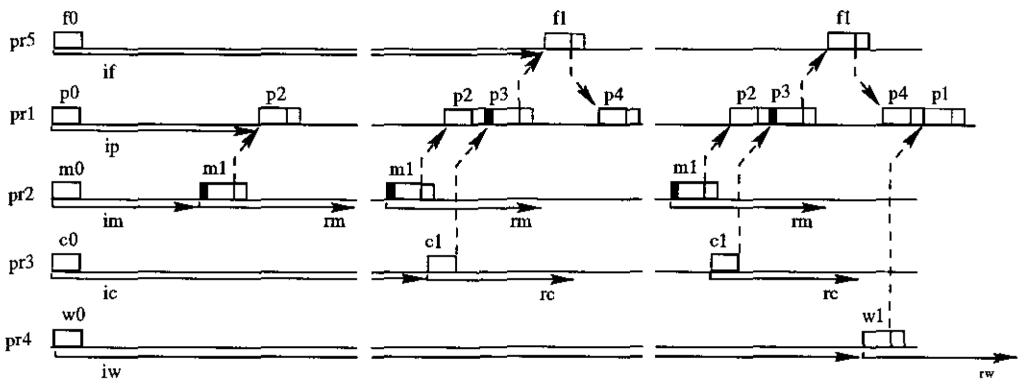

Figure 6: A schedule for the maximum parallellism case.

Pflow	$i_f \geq Tf0$ $r_f \geq Tf1$
Ppump	$i_{p2} \geq Tp0$ $lcd(r_m, r_c, r_w) \geq Tp2 + Tp3_0 + Tp3_1 + max(Mcf + Tf1_0 + Mf, Tp3_2) + Tp4 + Tp1$

All these conditions combined must be solvable for periods and off sets. To be able to persue a quantitative schedulability analysis we choose explicit numbers for the above mentioned durations in arbitrary units u. Changing this unit results in some scaling with regard to processor speed. The choosen durations are roughly proportional to the number of statements in each bead.

$Tm0 = 2u$	$Tm1_0 = 1u$	$Tm1_1 = 5u$	$Tm1_2 = 1u$	
$Tw0 = 1u$	$Tw1_0 = 1u$	$Tw1_1 = 1u$	$Tw2_0 = 1u$	$Tw2_1 = 1u$
$Tc0 = 0u$	$Tc1 = 2u$			
$Tf0 = 1u$	$Tf1_0 = 1u$	$Tf1_1 = 3u$		
$Tp0 = 6u$	$Tp1_0 = 6u$	$Tp1_1 = 1u$	$Tp2_0 = 5u$	$Tp2_1 = 1u$
$Tp3_0 = 2u$	$Tp3_1 = 2u$	$Tp3_2 = 1u$	$Tp4_0 = 2u$	$Tp4_1 = 1u$

The channel delay is taken to be $M = 2u$.

With the times as given in the specification (in ms) and the assumed execution times in execution units u the following conditions on the periods remain:

(1) $7u \leq r_m \leq 30 - 12u$
(2) $2u \leq r_w \leq 10000 - 11u$
(3) $4u \leq r_c \leq 100 - 9u$
(4) $25u \leq lcd(r_m, r_c, r_w)$

and the conditions on the off sets are:

(5) $2u \leq i_m \leq 30 - 13u$
(6) $u \leq i_w \leq 10,000 - 9u$
(7) $i_c \leq 100 - 15u$

Typically, the lower bounds stem from the limits on the processor loads, and the upper bounds from the critical path to deadline. The off-sets that are not mentioned here follow straightforwardly from the scheduling assumptions and are not subjected to additional constraints. The off-set matching, however, results in two more equations:

(8) $\exists x \in \mathbb{N} \cup \{0\} : i_c = i_m + 10u + x * lcd(r_m, r_c, r_w)$
(9) $\exists y \in \mathbb{N} \cup \{0\} : i_w = i_c + 10u + y * lcd(r_m, r_c, r_w)$

It is best to take common divider of the periods, as well as the periods themselves as large as possible, thus $r_m = lcd(r_m, r_c, r_w)$. From the constraints (1) and (4) we can compute the bound on the processor speed to make the described schedule feasible: $u \leq 30/37ms$. The other equations can also be solved for these values of u.

11.5 Off-line scheduling and minimum parallelism

In this case all processes are mapped on a single processor. Logical channels will require no physical communication channels and we assume that message passing costs no time ($M = 0$). Again we will construct a strictly periodic schedule. Since all blocks must be allocated to the single processor some tighter condition on the processor speed is expected. The schedule will be a merge of the maximum parallelism situation where we will complete all blocks of a transaction before an other transaction is carried out. Thus starting times will be:

- the block sequence $m0, c0, f0, p0, w0$ at $t = 0$;
- the sequence $m1, p2$ at $t = i_m + x * r_m$ with $x \in \mathbb{N} \cup \{0\}$;
- the sequence $c1, p3, f1, p4$ at $t = i_c + x * r_c$ with $x \in \mathbb{N} \cup \{0\}$;
- and the sequence $w1, p1$ or $w2, p1$ at $t = i_w + x * r_w$ with $x \in \mathbb{N} \cup \{0\}$.

The conditions on the off-sets and periods are now:

(1) $r_m \leq 30 - 11u$
(2) $r_w \leq 10000 - 10u$
(3) $r_c \leq 100 - 9u$
(4) $36u \leq lcd(r_m, r_c, r_w)$
(5) $10u \leq i_m \leq 30 - 12u$
(6) $i_w \leq 10,000 - 8u$
(7) $i_c \leq 100 - 13u$

Again, the lower bounds stem from the limits on the processor capacity, and the upper bounds from the critical path to deadline. The off-set matching results in following equations:

(8) $\exists x \in \mathbb{N} \cup \{0\} : i_c = i_m + 13u + x * lcd(r_m, r_c, r_w)$
(9) $\exists y \in \mathbb{N} \cup \{0\} : i_w = i_c + 14u + y * lcd(r_m, r_c, r_w)$

Again, the bound on u follows from equations (1) and (4), $u \leq 30/47ms$, and for these values the schedule is feasible.

12 On-line preemptive priority scheduling

In this case we employ a model similar to [7] in which processes are given an attribute *periodic* or *sporadic* and shared objects are present on which transactions are carried out. Since our programming model is simpler and does not contain the concept of objects we will introduce in addition to periodic and sporadic processes a third process class, *shared*, that has a very similar role as the shared objects in [7]. The execution of processes in this model goes as follows.

Periodic processes are each given a period. This period must be inferred from timing requirements. The periodic processes are timer triggered and released at the beginning of each period.

They are suspended after initialization and subsequently after each execution of the body of their **while** *true* **do** loop.

Sporadic and *shared* processes are message triggered, they become ready to run as soon as a message arrives. They are suspended when they attempt to receive a message from an empty buffer. The difference between sporadic and shared processes is that (usually) shared processes receive messages from several channels, originating from more than one process. Sporadic processes typically receive messages from a single channel originating from the environment. Sporadic and shared processes are treated differently in the schedulability analysis.

Periodic and sporadic processes have a number of attributes that have to be assigned during system generation: a priority P, a deadline D and a period T. Shared processes handle the message from the highest priority sender first, and they run at the priority of the highest priority sender that has a buffered or executing message. This mechanism is called priority inheritance and solves problems with priority inversion. Because a shared process must finish handling a message before a next message can be accepted, the progress of high priority processes can be blocked temporarily by the handling of a message from a lower priority process. To limit the blocking time to the minimum, the shared process runs at the priority of the mentioned highest priority sender.

To enable a schedulability analysis, worst-case execution times must be computed for the periodic and sporadic processes. For shared processes a separate worst-case execution time must be computed for the handling of messages out of each incoming channel.

The following table gives the attribute assignment for the mine-pump processes.

process	type	T_i	D_i	worst-case execution times
Pmethane	periodic	T_m	$dc - T_m$	$C_{methane}$
Pcheck	periodic	T_c	$df - T_c$	C_{check}
Pwater	sporadic	dlh	$min(dh, dl)$	C_{water}
Ppump	shared			$C_{pump,check}, C_{pump,water}, C_{pump,methane}$
Pflow	shared			

From the critical path analysis it follows that the specified deadline for response to a critical condition that must be met by a periodic processes (e.g. *df* for *Pcheck*) corresponds to the execution of a period after a measurement plus the response time after the next measurement. Thus, after each release only the deadline minus period, as indicated in the table, remains in the present model. The notation for the relevant worst case execution times are also indicated in the table. These entities must be computed using detailed knowledge of the platform. The execution times for the periodic and sporadic processes include the execution of a complete transaction that is initiated by these processes, so they include the invocation of the corresponding pump action. The indicated executions durations for message handling in the shared processes are required for computing the blocking-time for other transaction on the shared resource managed by them. Although, both the processes *Ppump* and *Pflow* are shared processes, *Pflow* only handles messages on behalve of *Ppump*, and for all schedulability considerations message handling time of *Pflow* can be incorporated into *Ppump*.

To analyze schedulability, the following recursion relation can be derived for the relevant response times along the lines of [7] (taking into account that *Ppump* is effectively the only shared process):

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j + \max_{j \in lp(i)} C_{pump,j}.$$

Here, the process-label i only runs over the periodic and sporadic processes:

$$i \in \{methane, check, water\}$$

and the sets $hp(i)$ and $lp(i)$ are the processes with a higher or lower priority than process i, respectively. If the recursion relation has a fixed point $R_i^{n+1} = R_i^n$ for some n, then the response time of process i is given by $R_i = R_i^n$. This response time must be smaller than the deadline D_i

for that process. This formula is only valid if the deadline for each process does not exceed the period.

To illustrate the use of this schedulabilty formula, we consider the numerical example. With a rate-monotonic priority assignment and worst-case execution times as given in section 11, we get the following relevant values to be plugged into this formula:

	C_i	$C_{pump,i}$	D_i	T_i	P_i
Pmethane	$13u$	$6u$	$30 - T_m$	T_m	3
Pcheck	$14u$	$12u$	$100 - T_c$	T_c	2
Pwater	$9u$	$7u$	$10,000$	$100,000$	1

Because *Pmethane* has the highest priority the equation for R_m can directly be solved:

$$R_m = 25u,\ T_m = 30(msec) - 25u \geq 25u$$

Taking the largest possible value of $u = 0.6\ msec$ and the largest possible value for T_m and T_c, the result of the analysis is:

$$R_m = 15\ msec,\ T_m = 15\ msec,$$

$$R_c = 28.2\ msec,\ T_c = 71.8\ msec$$

$$R_w = 29.4\ msec$$

This satisfies the deadlines as given in the table. With the present approximation the system is guaranteed schedulable for $u \leq 0.6$, but cannot be guaranteed if u is higher.

Note that this bound on u is slightly worse than the on found for the off-line scheduling on one processor. However, in the present analysis we have made some more conservative estimations on the location of measurements and responses in the program. Also some worst-case assumptions on possible blocking due to *Ppump* were made here. In the off-line scheduling approach such blocking can be avoided.

13 Discussion

13.1 Satisfying "good programming" requirements

In chapter 4 we have listed the "good programming" requirements that have to be satisfied by programming languages as published by Gligor and Luckenbaugh [10]. We will now discuss some of the typical features of our approach in the light of these requirements.

The language extension we have proposed includes only a small number of new primitives which provide expressiveness through the numerous ways they can be combined. However, the type of timing constraints that can be formulated is restricted by carefully chosen syntactical rules. For example, only events (execution instances of primitive statements) that are "causally connected" can have an explicit timing relation. (By "causally connected" we mean here the transitive closure of the sequential precedence of statement-executions and the message-send/receive relation.) Timing annotations do not introduce new causal precedences. This separation of timing and synchronization primitives is an intentional language design choice inspired by the orthogonality and "separation of concerns" requirements of [10].

1. Orthogonality: concurrent programming languages already contain primitives for synchronization. Duplication of functionality should be avoided.

2. Separation of concerns: synchronization and timing are treated as completely separate issues (synchronization must also be offered in languages without timing constraints). Primitives should not combine such independent aspects.

But is also introduced because of analyzability.

3. Analyzability: a timing constraints can be verified (statically or dynamically) by considering time bounds (upper and/or lower) on sequences of causally related actions. Whether the constraints are realized by employing synchronized clocks or by enforcing upper and lower bounds on actions connecting the events, either statically or dynamically, is unimportant at the language level. It is a system implementation issue.

In this paper we have demonstrated that the timing primitives can be formally defined and have a simple semantics, two other important requirements mentioned by [10]. More importantly, we have demonstrated how these formal definitions can be employed to prove correctness of programs.

13.2 Abstraction and refinement

Another requirement on approaches to real-time programming that we mentioned in sections 2 and 3, is support for abstraction and refinement of timing behavior. Our approach supports event abstraction. Event abstraction is extensively discussed in the context of distributed systems, particularly in distributed debugging. Many publications are available on the subject. For an interesting overview and references see [17]. With our present approach we have attempted to make an explicit connection between this field and real-time programming. However, in contrast with most work on timed events in distributed systems, where only logical time stamps are considered [18] that indicate the order of events, we attach *real-time* values to, possibly abstract, events. We will henceforth speak of *timed-event abstraction.* Although it is already present in our language, the scope and power of this form of abstraction becomes more manifest when procedures are introduced in our simple language.

We illustrate this with the following example. Our language contains an instruction to read form a device register, *in*(d, x). The purpose of such a read-instruction is to obtain information on the state of the environment. In our mine-pump example we have made the simplifying assumption that when a device register is read, the value obtained corresponds to that state at the execution moment of the instruction. This is too simple a view: in practice a value can only be obtained after the device has received notice that a measurement must be taken. Also, data will not be immediately available but some time is required by the device to collect and supply a value. Instead of performing a simple read action on a register, we sketch the following, more appropriate procedure that supplies both a value and a time, t at which this value holds. For this, we introduce timing parameters that may be supplied with a procedure. These parameters are similar to the timing parameters attached to the asynchronous messages.

```
//dev   = { d1,d2 }
//const = {indicate, epsilon, delta, scale, offset}

procedure read_device(var x, [?t])
begin
        out(d1, indicate) [?t1];
        in(d2,x) [>t1+delta ; t:=t1+epsilon];
        x := x*scale + offset
end
```

A request to collect a value must be written to the device in register $d1$. The device requires at least some time *delta* to obtain a value and make it available in register $d2$. The value supplied is actually taken from the environment at time *epsilon* after the request was indicated to the device. Both the value and this time, $t = t1 + epsilon$, are returned to the caller. We assume that *epsilon* and *delta* are constants that are characteristic for the device, and $0 < epsilon < delta$. Note that this procedure can transparently replace our naive time-annotated read action (where a single register, d, is sufficient).

```
        in(d,x) [?t];
```

Note, that all the device particulars are contained in this procedure (information hiding) and that platform dependent execution durations of statements are irrelevant in formulating this procedure. In our approach, abstraction amounts to hiding the detailed timing of a set of events to obtain a single, timed, abstract event. Procedures like the one above provide control over, or allow inspection of, execution moments of relevant (possibly abstract) events. The timing abstraction provided by the procedure in the example is similar to the assumed functional abstraction: the time of the moment of inspection of the environment as well as the obtained value require some device dependent adjustment to make them adhere to the abstraction level of the caller.

Also note, that in the example it is not useful to consider the start time and/or end time of the procedure *read_device*. They would only provide bounds on the moment of occurrence of the actually interesting event, and the strictness of these bounds depend on the execution speed of the platform as well as possible preemption of the procedure's execution.

The approach of refining abstract events contrasts with abstraction from execution durations often used in the literature. Consider the following procedure.

```
procedure f
begin
        g; h;
end
```

Here the execution duration of f is (larger than) the sum of the execution duration of g and h. In that sense, f abstract from the details of the duration of g and h separately. Event abstraction is more useful at the programming stage than duration abstraction for the following reasons.

- It is often not necessary and even undesirable to consider and formulate constraints on all execution durations of sub-procedures separately.
- If a procedure is used that contains no timed events, e.g. a procedure that implements a standard non-real-time algorithm, it can be incorporated as-is: no timing constructs need to be added.

13.3 The nature of timing parameters

Timing annotations and the timing parameters contained therein are programming concepts. They need not be present in the generated executable code. They are introduced to express the allowed executions of a program for as far as it concerns timing. The allowed execution trace that is ultimately selected by the platform and details of the selection procedure are not a concern of the programmer. It is therefore possible that timing parameters are not used or exchanged by processes at run-time. Instead, e.g. scheduling directives (priorities) may be used. Consider the mine pump example: statically it is verified that the timing constraints are satisfied and no explicit reference to them is required during run-time (no timing values need to be passed with the messages). This was true in the on-line as well as off-line scheduling case that we presented.

14 Comparison with other approaches

There are a number of survey papers on languages that can be applied to development of real-time systems [8, 11, 12]. From these studies Ada appears as the better languages for real-time programming. RT-Euclid is also a notable language. We will confine our comparison to these latter two languages and a couple of recent proposals that were not considered in the mentioned surveys.

In the next couple of subsections we briefly describe the features with regard to timing of Ada [1, 4], RT-Euclid [16], and the object oriented languages DROL [23], RTC++ [15], FLEX [20] and Sina (with real-time extensions) [3, 5]. After that we discuss the major differences with our approach.

14.1 ADA

The Ada language [4] was developed to supply the US Department of Defense with a programming language for embedded applications. Presently DoD contractors are required to use Ada. It has become an international (ISO and ANSI) standard and is widely used for the implementation of real-time software. Numerous changes have been incorporated in the language's current version Ada 95 [1].

The language supports modular concurrent programming through constructs like subprograms, packages and tasks. Packages and subprograms constitute reusable components with well defined interfaces. Concurrency is provided by tasks. Various synchronization and communication primitives are offered, like *rendezvous*, semaphores and shared variables. The introduction of a limited form of inheritance and dynamic binding in Ada 95 enable an object-oriented programming style. The real-time-systems annex contains mechanisms for dealing with priority inversion, for selecting scheduling policies, for task abortion, for dynamic priorities, and the like. Ada 95 also supports distribution.

We are mainly interested in how Ada deals with timing requirements. There are essentially three features in this regard:

- Facilities to define and manipulate task priorities
- A proper (since Ada 95) delay statement that can be used to release tasks at desired instances.
- A timed-entry-call with which a statement block can be abandoned after a certain time has expired.

The timed-entry-call is a method to decide on flow of execution on the basis of the amount of time that has expired. It appears that Ada does not provide means for explicitly specifying deadlines. Only release-times can be expressed using the delay statement. Process priorities are offered, but are indirect means for realizing deadlines. They are in some situations sufficient to give timeliness guarantees [7], but the timeliness requirements have to be recorded and maintained separately from the program. We have already mentioned the context dependence and the resulting reduced reusability of program components with hard-coded priorities.

Ada could be considerably improved if primitives were offered for specifying timing constraints and language implementation mechanisms were provided to (1) detect such constraints and (2) direct effort to realizing them.

Because of its intended wide applicability, Ada must offer sufficient flexibility to programmers. This resulted in a high number of primitives. Therefore, a formal definition of the language would be exceedingly difficult to generate and probably useless in any practical application.

The reader may be interested to compare the minepump problem as implemented in Ada 95 [7] with the solution presented in this paper. We observe the following features of this implementation that contrast with our version.

- Strictly periodic release-times for the periodic processes are specified in the Ada 95 program. This requires additional design choices, e.g. the constraint on the interval between successive measurements of the methane level must be divided into a period and a deadline. When this period is made smaller, the system load is increased, when it is made larger the available time to the deadline is tighter and the number of platforms that can execute the program is reduced. Thus, the programmer is enticed to take platform properties into consideration. In our approach such decisions are moved to the system generation phase and the program is kept more general.
- Although in the Ada 95 program actions are specified when certain deadlines are expired, the deadlines themselves are not syntactically recognizable. Timing variables are normal program variables and there are no syntactic restrictions on their use.
- The Ada tasks contain hard coded priorities and are therefore context dependent.

- Important deadlines have to be recorded outside the program for later use in the schedulability analysis. The authors provide a table.

The observed properties hamper reusability of components and reduce the number of execution platforms that can be used. It is interesting to note that timing parameters (as regular parameters of procedures) are used in the Ada program and the previously mentioned event abstraction is employed to a certain extent.

14.2 Real-Time Euclid

Real-time Euclid [16] is a language derived from Pascal. It supports modularity and concurrency, but it does not support distribution. The authors address through their work on RT-Euclid issues on real-time and reliability and they do this in a very straightforward and coherent manner. The language is intended for the hard-real-time domain where it is necessary to provide timeliness guarantees at all costs and provide a high degree of reliability. In that light, language features are offered to (1) address hard-real-time requirements and (2) address reliability issues. The most important features falling in the first category are:

- Timing constraints. Frames are attached to processes. A frame is an activation window available for executing the process. Time constraints can be specified as attributes of such frames.
- Language restrictions to guarantee termination of all statements and to limit resource usage. In particular there is no dynamic process creation, there are no dynamic data structures, there is no recursion, and iterations and synchronizations are time bound.
- Program structure to enable schedulability analysis. In particular there is an initialization section to each module, which is executed before (often cyclic) execution of processes defined in the module, starts.

A feature in the second category is:

- Exception handling.

Although RT Euclid seems simple enough to provide a formal definition of language primitives we have not been able to find it in the literature.

We like to mention that it is not necessary to address limitations on execution time and resource usage by introducing language primitives. Programs in Real-Time Euclid are guaranteed not to exceed certain time bounds but this may be at the expense of functional correctness. e.g. a repetition that is abandoned too early because the limit on the number of iterations has been exceeded may yield an unpredictable and undesired result. Program correctness requires *both* functional correctness and timeliness. It must be established that a repetition yields an acceptable result *and* that it terminates timely. Therefore, an explicit bound on the number of iterations through a repetitions should only indicate before when the repetition is guaranteed to arrive at the correct result, and it should have no implications at run-time. The only reason to have the programmer specify it is that such information is usually hard to deduce at compile time but is required to perform the timing analysis.

14.3 DROL

DROL [23] is a programming language for distributed object-oriented real-time systems with a run-time system that is implemented on the ARTS kernel. It obtains the facilities to implement a distributed system from this kernel. There are a couple of problems in the realm of real-time object-oriented computing which the authors want to specifically address in their work:

- Flexibility: The authors aim at the implementation of systems that can handle highly dynamical environments.

- Proper handling of exceptions: They particularly emphasize the necessity to check for timing failures at both the client as well as the server side. This is required in case communication is not reliable or not time-bounded.
- Specification of timing requirements: They argue that it is not possible to encapsulate timing information in a single object because in a distributed environment timing constraints must be met through the combined effort of objects at different sites.

The DRO programming model is an extension of C++. The extensions are in the areas of distribution and concurrency, concurrency control, real-time and exception handling.

In addition to C++ objects, DRObject are introduced which are large grained objects that can be distributed. DRObjects consists of one base object, that implements the functional behavior and two meta-level objects that govern the communication and synchronization between DRObjects. An object's public member functions can be called by other objects. In addition, it is possible to define "active" member functions which are periodically invoked automatically. A DRObject does not handle more than one invocation concurrently. The communication protocol can be shaped by the programmer to his needs using the two meta-level objects, e.g. by specifying synchronous, and asynchronous invocation handling, reliable communication, handling of timing exceptions and commitment or abortion of transactions at the meta-level.

There are various constructs that have been introduced to enable specification of timing properties of the DRObjects. e.g., active members can be given a period and deadline, member functions of DRObjects can be given a worst-case execution time within which an invocation is likely to finish without raising exceptions and clients may specify the available time for an operation. Summarizing, the following types of timing constraints are offered:

- duration constraints on object invocations,
- periods and deadlines of active members,
- the definition of timing exceptions through timeout specification and
- time-polymorphism.

Time polymorphism is a feature that allows the definition of alternative degrees of service with different worst-case time consumption and a dynamic selection of this degree based on the available time.

DROL is based on the principle of best effort and least suffering. Timing constraints are not guaranteed. The mechanisms that are employed to achieve as best a timing behavior as possible are twofold.

- Scheduling and time polymorphism. Each object invocation is dynamically scheduled and dynamically bound to the implementation that will best meet the available time as provided by the client. (These mechanisms achieve what the authors call "best effort").
- Handling of timing failures. It can be specified how timing exceptions should be handled. Thus, propagation of timing failures through the system can be avoided. (This achieves what the authors call "least suffering").

The DRO Language is rather complex. A formal definition of the semantics is likely very hard to provide. However, according to the authors DROL is very well suited to deal with very dynamic systems. It is flexible with regard to synchronization and exception handling. However, the "best effort" approach can not result in guarantees on timing behavior. Composability and abstraction of timing behavior of DRObjects is not explicitly addressed.

14.4 RTC++

RTC++ [15] is also implemented on the ARTS kernel. The ARTS/RTC++ platform is an environment for the development of distributed object oriented real-time systems. It differs form the DROL language in that it focuses almost exclusively on the issues that are directly addressed by the kernel. It has a lesser complexity and abstraction level then DROL. (e.g. distribution and the problems related to reliability and timing are not addressed by the authors.) The ARTS object model and issues dealt with by this kernel are reflected in the language and its primitives.

The ARTS kernel provides a multi-threaded object model (intra-object concurrency), enables remote object invocations with timing constraints and offers mechanisms to deal with the problem of priority inversion. Object-invocations are only synchronous (client waits for completion). Invocations to an object can be given priorities to influence the order in which they are handled. The timing constraints on invocations are managed through the so called "time-fence" protocol. In RTC++ some linguistic handles to these kernel facilities are offered. The ARTS kernel schedules an application on-line (no off-line scheduling) and uses rate-monotonic scheduling for HRT threads. The RTC++ program must be analyzed on its schedulability and is designed to suit this approach. This emphasis on schedulability analysis explains the relative importance that is given to priority inversion in the ARTS/RTC++ literature.

As the name suggests RTC++ is a real-time extension of C++. Typically, a distinction between real-time objects and normal C++ objects is made. The real-time objects are active and are mapped on light-weight concurrent ARTS run-time objects. These ARTS objects are the units of distribution. Real-time objects may have timing constraints and the object activities may be given priorities. We can distinguish the following principle real-time features in the language:

- Real-time objects with master (periodic) and slave (call handling) threads. Slave threads inherit the priority from the caller.
- Constraints on periodic threads (e.g. period and deadline).
- Constraints on the duration of (member) functions.
- Timing constraint statement blocks.
- Exception handling mechanism for timing violations.

The division of some constraints into periods and deadlines for periodic threads is required when this language is used. We already commented on this in our discussion of the Ada language.

The ARTS kernel employs rate monotonic scheduling of HRT tasks and the schedulability analysis is based on this scheduling regime. In this way it is possible to guarantee timeliness for certain applications.

14.5 FLEX

FLEX [20] is a real-time language developed in the Concord project at the University of Illinois. It focuses on programming of systems that can operate in very dynamic environments. In this sense it aims at similar applications as the DROL language. It is derived form C++ and obtains object-oriented features from this language. FLEX does not have primitives for distribution or parallel programming. Hence the only additions to the C++ language pertain to timeliness. In order to express timing constraints and satisfy them as best as possible a number of primitives and mechanisms are offered in FLEX:

- constrained blocks,
- imprecise computation,
- performance (time) polymorphism.

Imprecise computation and performance polymorphism are techniques to exchange precision or resource usage for time. These techniques are also encountered in other approaches and are not interesting in the present context. We will not further discuss them.

Timing and resource constraints can be formulated in FLEX using a constrained block construct. This language construct enables the definition of constraints on a block of statements together with an exception handler that is invoked when, at run-time, the constraint is not satisfied. Various attributes of such blocks can be employed to formulate the constraints: start, finish, duration, periods and priority. Attributes of other blocks can be used in the formulation of a constraint imposed on a particular block. There are two possible ways to successfully execute a constrained block: by satisfying the constraint or by calling the exception handler. This leads to ambiguities on semantics which the authors identify: when a constraint is not satisfied either an exception handler can be called or the execution can be suspended until the constraint becomes satisfied. However, this ambiguity is solved in a rather confusing manner by letting this choice depend on subtle differences in notation. We are not aware of any formal definition of the constraint block.

Although the authors state that there are no provisions for concurrency in FLEX, they provide a couple of examples that only make sense in a concurrent system. We see, however, two major problems with the proposal when concurrency is used. (1) It is not clear what the scope rules for block attributes are or should be. (2) There may be several invocation instances of a block, each with different values for the attributes. The block label is not sufficient to distinguish between such instances and constraints may be ambiguous.

14.6 Sina

Sina [2] is an object oriented, concurrent programming language, suitable for distribution. Object are essentially abstract data types that hide implementation. With the use of so called *composition filters* typical object oriented mechanisms like inheritance and delegation are realized. Sina offers both inter- as well as intra-object concurrency. Inter-object messages are either directly handled or queued in a, conceptually infinite, message queue of the receiving object. The order and way in which messages are handled is expressed in the filter construct. Real-time filters for Sina are introduced in [3, 5], they are used to express timing constraints on invocation (message) acceptance and completion. Filters incorporate the following features for timing purposes.

- Timing constraints can be attached to messages. For every message an earliest start time can be specified, a deadline, after which it should not be serviced, and a period. Such times can be specified absolutely, or relative to the moment the message passes the first filter. (There is a sequence of filters that a message must pass before acceptance.)
- Real-time filters can manipulate message constraints. In particular, new constraints may be added to the ones already specified in the message.
- Filters can specify policies for dispatching queued messages. Messages are scheduled according to the applicable constraints.
- Timed-out messages can be removed from the queue. In hard-real-time systems it often does not make sense to execute a message for which the execution deadline has expired.

In addition to real-time filters it is possible to specify so called time frames for the execution of (periodic) tasks. Sina is the only language for which we have found a formally defined semantics.

14.7 Comparison

A distinction can be made between real-time systems that must cope with very dynamic environments, where in general no guarantee can be given that timing requirements are satisfied in all circumstances, and systems that operate in environments that are predictable enough to be able to analyze all possible behaviors and provide timeliness guarantees. Although we believe that a real-time programming language should in principle not be tailored to the application domain in this sense, it is customary to make the distinction at the present. DROL, FLEX and Sina focus on very dynamic systems. Real-time Euclid and RTC++ on predictable, provably correct systems. Ada has been applied to both regimes. All the languages contain features that are valuable in the intended application domain.

We note the following differences between our approach and the ones mentioned above.

1. Timing constraints are not attached to program blocks.
   All languages that we considered, except for Ada which does not support constraints, formulate timing constraints using either constrained blocks, or constrained method/procedure calls or both. Constraints on procedures are usually on start and end times and very similar to the block constraints. An advantage is, that constraints formulated in this way can be easily translated into scheduling directives for the block(s) of statements they are attached to. It also has the advantage that timing violations during execution of a block can be handled by a specified exception handler. (No language, however, provides desired features such as atomicity of constrained blocks under timing failures.) We have intentionally departed from formulating timing constraints (like durations, deadlines and earliest start times) on blocks because they do not directly pertain to state transitions brought about by the computation. In fact, block constraints specify restrictions on processor allocations, which only provide bounds on the relevant state transitions. The accuracy of such bounds depend on typical implementation issues like on whether such blocks are preemptable or not. When execution moments of primitive statements are used, one can specify constraints on any state transition effectuated by the program in an accurate and simple way. Also, block attributes and constraints on them have no direct relation with externally observable events. A system specification most often refers to such events. In our approach there is a direct relation between the control requirements (expressing external conditions) and the annotations. Considering execution moments of statements combines well with event abstraction.

2. Explicit scope rules for timing parameters.
   Often, constraints need to be specified on blocks that are not in each others scope. Also previous execution instances off blocks must sometimes be considered. Scope rules and parameter passing mechanisms must be specified for block attributes. Whether or not such rules and mechanisms are at all present in the considered languages is not made sufficiently clear in the respective publications. Also over-specifications are introduced in the provided examples that makes the absence of scope rules less urgent (e.g. splitting a single constraint on the response of a periodic process in a period and a deadline within the period.

3. Restricted syntax of timing-constraint expressions.
   In particular we have no data dependencies of timing variables. None of the other methods contains syntactic restrictions that disable data dependent constraints. Such constraints are most often impossible to verify without very detailed knowledge of the environment.

4. Formulation of timing requirements in an end-to-end fashion.
   Usually timing requirements pertain to the control of observable events. In our approach such timing constraints can be formulated without constraining unnecessarily sub-actions carried out by the thread of execution connecting the events. Event abstraction is essential for this. Although in other approaches event abstraction can probably also be used, we have not found any systematic application of the principle.

5. Formal definition of timing constructs.
   Hardly any of the investigated papers provided a formal or very precise description of syntax and semantics. This makes it very difficult to understand the full extent and implications of the various approaches.

14.8 The combination of real time with object orientation

With their work the authors of Sina advocate composition filters as a solution to the well known concurrency anomalies [21] in concurrent object-oriented languages . They also claim that some inheritance anomalies exist with regard to timing specification that are resolved using real-time filters [3, 5]. The real-time anomalies result in a reduced reusability of code of classes under inheritance. They claim that the specification of timing restrictions of classes result in a need to redefine unnecessarily many methods of a superclass, or that the change of the method of a class results in the need to redefine many methods of super- as well as subclasses. They indicate the following causes for real-time inheritance anomalies:

- the definition of real-time constraints in method bodies,
- the specification of timing constraints that cannot be inherited separately from other features of a class,
- the inability to formulate a constraint that pertains to all (or many) methods of a class.

We believe that such real-time anomalies are absent, or at least not problematic, if timed-event abstraction is used. This is mainly so because only those parts of a constraint need to be specified in a method that are characteristic for the abstract event as offered by the object. Superclasses can offer control over (abstract) events that are not affected by added functionality in sub-classes. Also refraining from over-specification of timing behavior helps to avoid the anomalies.

In [22] an object-oriented programming model is introduced in which timing constraints between invocations of an object or group of objects can be specified by the programmer in separate entities, called synchronizers. It was deemed necessary to separate timing and synchronization constraints from the functional part of a program to obtain flexibility and composability. We have attempted to show that when event abstraction is employed, timing constraints can be formulated without any reduction in composability compared to non-timed programs: a server provides abstraction of and control over one, or several, execution event(s) and the caller determines the constraints on the event(s) if he requires any. This provides maximum flexibility and context independence. In our opinion timing and synchronization must be treated as separate issues.

15 Conclusion

We have described an extension for programming languages for including timing constraints. The introduced syntax was just a vehicle to illustrate the approach and details of the syntax and semantics are not essential. They may still depend on the particular application domain for which a real-time language is employed, e.g. hard/soft-real-time systems, dynamic/predictable environments. We believe, however, that the following properties *are* essential for any successful approach to real-time programming.

- Formal techniques are essential, at least to define a real-time programming paradigm. Many current proposals are difficult to understand and apply because a formal semantics of new constructs is not provided.
- A strict separation of concerns in the programming and system generation phases should be used. All platform and context dependence should be addressed during system generation. In this regard, the only difference with non-real-time programming is, that the system generator may decide that there is no implementation of the program on the specified platform.
- Real-time programs *can* and *should* be designed *and verified* independently of an execution platform.
- The proper way of dealing with abstraction and refinement of timing behavior in real-time programs is, what we called *timed-event abstraction*.

- Over-specification of timing behavior should be avoided. It is usually done on the basis of platform considerations.

References

[1] "Ada 95 Reference Manual", *International Standard ANSI/ISO/IEC-8652:1995*, January 1995.

[2] M. Aksit, "On the design of the object-oriented language Sina", Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 1989.

[3] M. Aksit, J. Bosch, W. v.d. Sterren and L. Bergmans, "Real-Time Specification Inheritance Anomalies and Real-Time Filters", Proc. of the ECOOP '94 Conference, LNCS 821, Springer Verlag, pp 386-407, 1994.

[4] G. Booch, "Software Engineering with Ada", Benjamin Cummings Publishing Company, Menlo Park CA, 1983.

[5] L. Bergmans and M. Aksit, "Composing Synchronization and Real-Time Constraints", ftp://ftp.cs.utwente.nl/pub/doc/TRESE.

[6] A. Burns and A.M. Lister, "A Framework for Building Dependable Systems", The Computer Journal, Vol. 34, No. 2, pp 173-181, 1991.

[7] A. Burns and A. Wellings, "Advanced Fixed Priority Scheduling", in "Real-time Systems; Specification, Verification and Analysis" edited by M. Joseph, pp 32-96, Prentice Hall, London, 1996.

[8] A. Burns and A. Wellings, "Real-Time Systems and their Programming Languages", Addison-Wesley, 1989.

[9] E.W. Dijkstra, "Notes on Structured Programming", Structured Programming, A.P.I.C. Studies in Data Processing No. 8, pp 1-81, Academic Press, New York, 1972.

[10] V.D. Gligor and G.L. Luckenbaugh, "An Assessment of the Real-time Requirements for Programming Environments and Languages", Proceedings of the 1983 Real-Time Systems Symposium, IEEE Computer Society Press, pp 3-19, 1983.

[11] W.A. Halang and A.D. Stoyenko, "Comparative Evaluation of High-Level Real-Time Programming Languages", International Journal of Time-Critical Computing Systems, Vol. 2, No 4, pp 365-382, 1990.

[12] W.A. Halang and A.D. Stoyenko, "Constructing Predictable Real-Time Systems", Kluwer Academic Publishers, Dordrecht-Hingham, 1991.

[13] D.K. Hammer *et al.*, "Dedos: A Distributed Real-Time Environment", Parallel & Distributed Technology, IEEE Computer Society, Winter 1994.

[14] J. Hooman, "Extending Hoare Logic to Real-Time", Formal Aspects of Computing, 6A: 801-825, 1994.

[15] Y. Ishikawa, H. Tokuda, and C.W. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints", Carnegie Mellon, 1990.

[16] E. Kligerman and A.D. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, 1986

[17] T. Kunz, "Event Abstraction: Some Definitions and Theorems". Technical Report TI-1/93, Technische Hochschule Darmstadt, Fachbereich Informatik, Darmstadt, Germany, February 1993, ftp://ftp.th-darmstadt.de/pub/docs/tech-reports/fb20/iti/at/THD-AT-1993-01.ps.Z

[18] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Vol. 21, No 7, pp203-217, 1978.

[19] I. Lee and V. Gehlot, "Language Constructs for Distributed Real-Time Programming", Proceedings of the 1985 Real-Time Systems Symposium, pp 57-66, IEEE Computer Society Press, 1985.

[20] K. Lin, J.W.S. Liu, K.B. Kenny and S. Natarajan, "FLEX: A Language for Programming Flexible Real-Time Systems", in M. van Tilborg and G.M. Koob(eds), Foundation of Real-Time Computing (Formal Specifications and Methods) pp 251-289, Kluwer Academic Publishers, 1991.

[21] S. Matsuoka, K. Taura and Y. Yonezawa, "Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages", in Proceedings OOPSLA '93, Sigplan Notices Vol. 28, No. 10, pp 109-36, 1993.

[22] S. Ren, G. A. Agha and M. Saito, "A modular Approach for Programming Distributed Real-Time Systems", to appear in Journal of Parallel and Distributed Computing, preprint 1996.

[23] K. Takashio and M. Tokoro, "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", Proceedings of the OOPSLA '92 Conference, ACM SIGPLAN Notices, Vol. 27, No. 10, pp276-294, 1992.

[24] G. Yu and L.R. Welch, "Program Dependence Analysis for Concurrency Exploitation in Programs Composed of Abstract Data Type Modules", Proceedings of the IEEE Symposium on Parallel and Distributed Computing, pp 66-73, Oct. 1994.

[25] N. Wirth, "Programming Development by Stepwise refinement", Comm. of ACM, Vol. 14, No. 4, pp 221-227, 1971.

[26] N. Wirth, "Towards a Discipline of Real-Time Programming", Comm. of ACM, Vol. 20, No. 8, pp 577-583, 1977.

A Proof of the water-level control program

In this appendix we prove that the program from section 5.3 satisfies the specification provided there. We repeat the program using m_i for execution moments in the program, to avoid confusion with timing variables in the specification:

```
P:
[ $m_1 := 0$ ];
while $true$ do [ $m_2 := m_1$ ];
              $in$(water, $x$)[ $< m_2 + d, ?m_1$ ];
              if high $< x$ then $out$(pump, $on$)[ $< m_2 + d$ ]
               else if $x <$ low then $out$(pump, $off$)[ $< m_2 + d$ ] fi fi
         od
```

By symmetry, only CC_1, CC_2', and CC_3' are proven, assuming the proof of CC_4' and CC_5' proceeds similarly.

A.1 Prove of CC_1

To prove CC_1, that is,

$$\forall t\, \exists t_1, t_2 : t_1 \leq t \leq t_2 \leq t + d \wedge readperiod(t_1, t_2)$$

where

$$readperiod(t_1, t_2) \leftrightarrow t_1 < t_2 \wedge read(\mathsf{water})\ \mathbf{at}\ t_2 \wedge (\neg read(\mathsf{water}))\ \mathbf{during}\ (t_1, t_2)$$

Define

$$CC_1(t) \equiv \exists t_1, t_2 : t_1 \leq t \leq t_2 \leq t + d \wedge readperiod(t_1, t_2)$$

The procedure followed will entail a prove for $CC_1(t)$ for unbound increasing values of t. Starting with $\forall t < 0 : CC_1(t)$ which is satisfied trivially because the domain of t are is formed by the non-negative reals, we arrive at $\forall t : CC_1(t)$ which is equivalent to CC_1

We assume initially $em = 0$, representing the assumption that nothing is executed before this program. The proof is based on the invariant

$I_1 \equiv term \wedge m_1 \leq em \leq m_1 + d \wedge \neg read(\mathsf{water})\ \mathbf{during}\ (m_1, em) \wedge \forall t < m_1 : CC_1(t)$

and the auxiliary assertions

$A_1 \equiv term \wedge m_2 \leq em \wedge \neg read(\mathsf{water})\ \mathbf{during}\ (m_2, em) \wedge \forall t < m_2 : CC_t(t)$

$B_1 \equiv term \wedge m_2 < em \wedge \neg read(\mathsf{water})\ \mathbf{during}\ (m_2, em) \wedge$
$read(\mathsf{water})\ \mathbf{at}\ em \wedge \forall t < m_2 : CC_1(t)$

$C_1 \equiv term \wedge m_2 < em < m_2 + d \wedge m_1 = em \wedge \neg read(\mathsf{water})\ \mathbf{during}\ (m_2, em) \wedge$
$read(\mathsf{water})\ \mathbf{at}\ m_1 \wedge \forall t < m_2 : CC_1(t)$

$D_1 \equiv term \wedge m_2 < m_1 < em \wedge \neg read(\mathsf{water})\ \mathbf{during}\ (m_2, em) \wedge$
$read(\mathsf{water})\ \mathbf{at}\ m_1 \wedge \forall t < m_2 : CC_1(t)$

$E_1 \equiv term \wedge m_2 < m_1 < em < m_2 + d \wedge \neg read(\mathsf{water})\ \mathbf{during}\ (m_2, em) \wedge$
$read(\mathsf{water})\ \mathbf{at}\ m_1 \wedge \forall t < m_2 : CC_1(t)$

Motivated by the timing annotation introduction rule, the proof of $\langle\!\langle A \rangle\!\rangle\ S[\,TA\,]\ \langle\!\langle C \rangle\!\rangle$ is represented by the proof outline $\langle\!\langle A \rangle\!\rangle\ S \langle\!\langle B \rangle\!\rangle\ TA\ \langle\!\langle C \rangle\!\rangle$.

The above assertions are used in the following proof outline.

```
⟪$term \wedge em = 0$⟫
[ $m_1 := 0$ ] ;
⟪$term \wedge m_1 = em = 0$⟫
while true do ⟪$I_1$⟫
                [ $m_2 := m_1$ ] ;
                ⟪$A_1$⟫
                $in$(water, $x$) ⟪$B_1$⟫ [ $< m_2 + d, ?m_1$ ] ; ⟪$C_1$⟫
                if high $< x$ then ⟪$C_1$⟫ $out$(pump, $on$) ⟪$D_1$⟫ [ $< m_2 + d$ ] ⟪$E_1$⟫
                 else if $x <$ low then ⟪$C_1$⟫ $out$(pump, $off$) ⟪$D_1$⟫ [ $< m_2 + d$ ] ⟪$E_1$⟫ fi fi
                ⟪$I_1$⟫
            od
⟪$CC_1$⟫
```

There are two less trivial steps in this outline, namely (1) from E_1 to I_1 and (2) from I_1 to CC_1:

Step 1
We show that $E_1 \rightarrow I_1$. Assume $term \wedge m_2 \leq m_1 < em < m_2 + d \wedge \neg read$(water) **during** $(m_2, em) \wedge$ $read$(water) **at** $m_1 \wedge \forall t < m_2 : CC_1(t)$. Note that $m_1 \leq em < m_2 + d \leq m_1 + d$ and that $m_2 \leq m_1$ leads to $\neg read$(water) **during** (m_1, em).

It remains to prove $\forall t \in [m_2, m_1) : CC_1(t)$. Consider $t \in [m_2, m_1)$. Let $t_1 = m_2$ and $t_2 = m_1$. Then $t_1 \leq t \leq t_2 \leq t + d$, since $m_2 \leq t \leq m_1 < m_2 + d \leq t + d$. We have $m_2 < m_1$, $read$(water) **at** m_1, and $(\neg read$(water)$)$ **during** (m_2, m_1).

Since $C_1 \rightarrow E_1$, also $C_1 \rightarrow I_1$.

Step 2
The last step, the proof of CC_1, follows from the "While True" rule, since $I_1 \wedge \neg term$ is equivalent to $false$ and $\forall t_1\, \exists t_2 > t_1 : I_1[t_2/em]$ implies $\forall t_1\, \exists t_2 > t_1\; t_2 \leq m_1 + d \wedge \forall t < m_1 : CC_1(t)$.

We show that this leads to $\forall t_0 : CC_1(t_0)$. Consider a point of time t_0. Let $t_1 = t_0 + d$. Then there exists a $t_2 > t_1$ such that $t_2 \leq m_1 + d$, and thus $m_1 \geq t_2 - d > t_1 - d = t_0$. I.e., $t_0 < m_1$ which leads to $CC_1(t_0)$.

Finally note that $\forall t_0 : CC_1(t_0)$ implies CC_1.

A.2 Prove of CC_2'

Next we prove CC_2', that is,

$\forall t_1, t_2 : readperiod(t_1, t_2, \text{high}) \rightarrow set(\text{pump}, on)$ **in** $[t_2, t_1 + d]$.

Define

$CC_2'(t_2) \equiv \forall t_1 : readperiod(t_1, t_2, \text{high}) \rightarrow set(\text{pump}, on)$ **in** $[t_2, t_1 + d]$.

Then $CC_2 \equiv \forall t_2 : CC_2'(t_2)$. Note that $CC_2'(t)$ holds at any point t where no read action takes place. The prove proceeds similarly to the one for CC_1. Let

$I_2 \equiv term \wedge \neg read$(water) **during** $(m_1, em) \wedge (m_1 = 0 \vee read$(water) **at** $m_1) \wedge \forall t < em : CC_2'(t)$

and the auxiliary assertions

$A_2 \equiv term \wedge \neg read$(water) **during** $(m_2, em) \wedge (m_2 = 0 \vee read$(water) **at** $m_2) \wedge \forall t < em : CC_2'(t)$

$B_2 \equiv term \wedge \neg read$(water) **during** $(m_2, em) \wedge (m_2 = 0 \vee read$(water) **at** $m_2) \wedge$
$read$(water, high) **at** $em \wedge \forall t < em : CC_2'(t)$

$C_2 \equiv term \wedge m_1 = em \wedge \neg read$(water) **during** $(m_2, em) \wedge (m_2 = 0 \vee read$(water) **at** $m_2) \wedge$

$\quad read(\textsf{water}, x)$ **at** $em \wedge\ \forall t < em : CC_2'(t)$

$D_2 \equiv term \wedge m_1 < em \wedge\ \neg read(\textsf{water})$ **during** $(m_2, em) \wedge\ (m_2 = 0 \vee read(\textsf{water})$ **at** $m_2) \wedge$
$\quad read(\textsf{water}, \textsf{high})$ **at** $m_1 \wedge\ set(\textsf{pump}, on)$ **at** $em \wedge$
$\quad \forall t < m_1 : CC_2'(t) \wedge\ \forall t, m_1 < t \leq em : CC_2'(t)$

which are used in the following proof outline.

```
⟪term ∧ em = 0⟫
[ m₁ := 0 ] ;
⟪term ∧ m₁ = em = 0⟫
while true do ⟪I₂⟫
              [ m₂ := m₁ ] ;
              ⟪A₂⟫
              in(water, x) ⟪B₂⟫ [ < m₂ + d, ?m₁ ] ; ⟪C₂⟫
              if high < x then ⟪C₂⟫ out(pump, on) ⟪D₂⟫ [ < m₂ + d ] ⟪I₂⟫
               else if x < low then ⟪I₂⟫ out(pump, off) ⟪I₂⟫ [ < m₂ + d ] ⟪I₂⟫ fi fi
              ⟪I₂⟫
         od
⟪CC′₂⟫
```

Again there are two less trivial steps in the prove.

Step 1
$D_2 \wedge em < m_2 + d$ leads to I_2.

Step 2
The last step, the proof of CC_2, follows from the "While True" rule, since $I_2 \wedge \neg term$ is equivalent to $false$ and $\forall t_1\ \exists t_2 > t_1 : I_2[t_2/em]$ implies $\forall t_1\ \exists t_2 > t_1\ \forall t < t_2 : CC_2'(t)$. Hence $\forall t_1 : CC_2'(t_1)$, thus CC_2'.

A.3 Prove of CC_3'

Finally we prove CC_3', that is,

$$\forall t : set(\textsf{pump}, off) \textbf{ at } t \rightarrow \exists t_1, t_2 : t \in [t_2, t_1 + d] \wedge readperiod(t_1, t_2, \textsf{low}) \wedge (\neg read(\textsf{water})) \textbf{ during } (t_2, t].$$

Define

$$CC_3'(t) \equiv set(\textsf{pump}, off) \textbf{ at } t \rightarrow \exists t_1, t_2 : t \in [t_2, t_1 + d] \wedge readperiod(t_1, t_2, \textsf{low}) \wedge (\neg read(\textsf{water})) \textbf{ during } (t_2, t].$$

Let

$I_3 \equiv term \wedge\ \neg read(\textsf{water})$ **during** $(m_1, em) \wedge \forall t < em : CC_3'(t)$

and the auxiliary assertions

$A_3 \equiv term \wedge\ \neg read(\textsf{water})$ **during** $(m_2, em) \wedge \forall t < em : CC_3'(t)$

$B_3 \equiv term \wedge\ \neg read(\textsf{water})$ **during** $(m_2, em) \wedge read(\textsf{water}, x)$ **at** $em \wedge\ \forall t < em : CC_3'(t)$

$C_3 \equiv term \wedge\ m_1 = em \wedge\ \neg read(\textsf{water})$ **during** $(m_2, em) \wedge read(\textsf{water}, x)$ **at** $em \wedge$
$\quad \forall t < em : CC_3'(t)$

$D_3 \equiv term \wedge m_1 < em \wedge\ \neg read(\textsf{water})$ **during** $(m_2, em) \wedge read(\textsf{water}, \textsf{low})$ **at** $m_1 \wedge$
$\quad \forall t < em : CC_3'(t)$

$E_3 \equiv term \wedge m_1 < em \wedge\ \neg read(\textsf{water})$ **during** $(m_2, em) \wedge read(\textsf{water}, \textsf{low})$ **at** $m_1 \wedge$

$$set(\mathsf{pump}, \mathit{off})\ \mathbf{at}\ em \wedge\ \forall t < em : CC_3'(t)$$

which are used in the following proof outline.

```
$\langle\langle term \wedge em = 0 \rangle\rangle$
$[\, m_1 := 0 \,]$ ;
$\langle\langle term \wedge m_1 = em = 0 \rangle\rangle$
**while** $true$ **do** $\langle\langle I_3 \rangle\rangle$
                   $[\, m_2 := m_1 \,]$ ;
                   $\langle\langle A_3 \rangle\rangle$
                   $in(\mathsf{water}, x)$ ; $\langle\langle B_3 \rangle\rangle$ $[\, < m_2 + d, ?m_1 \,]$ ; $\langle\langle C_3 \rangle\rangle$
                   **if** high $< x$ **then** $\langle\langle I_3 \rangle\rangle$ $out(\mathsf{pump}, on)$ $\langle\langle I_3 \rangle\rangle$ $[\, < m_2 + d \,]$ $\langle\langle I_3 \rangle\rangle$
                    **else if** $x <$ low **then** $\langle\langle D_3 \rangle\rangle$ $out(\mathsf{pump}, \mathit{off})$ $\langle\langle E_3 \rangle\rangle$ $[\, < m_2 + d \,]$ $\langle\langle I_3 \rangle\rangle$ **fi fi**
                   $\langle\langle I_3 \rangle\rangle$
          **od**
$\langle\langle CC_3' \rangle\rangle$
```

The last step, the proof of CC_3', follows from the "While True" rule, since $I_3 \wedge \neg term$ is equivalent to $false$ and $\forall t_1\ \exists t_2 > t_1 : I_3[t_2/em]$ implies $\forall t_1\ \exists t_2 > t_1 \forall t < t_2 : CC_3'(t)$. Thus $\forall t_1 : CC_3'(t_1)$, that is, CC_3'.

Computing Science Reports **Department of Mathematics and Computing Science**
Eindhoven University of Technology

In this series appeared:

93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness, p. 24
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Komatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\!\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.
94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.

94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma J. Davies R. Gerth S. Graf W. Janssen B. Jonsson S. Katz G. Lowe M. Poel A. Pnueli C. Rump J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef	Concrete process algebra, p. 134.
95/04	J. Hidders	An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra	Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber	Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15
95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement

		Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.
95/27	P.D.V. van der Stok, J. van der Wal	Proceedings of the Real-Time Database Workshop, p. 106.
95/28	W. Fokkink, C. Verhoef	A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
95/29	H. Jurjus	On Nesting of a Nonmonotonic Conditional, p. 14
95/30	J. Hidders, C. Hoskens, J. Paredaens	The Formal Model of a Pattern Browsing Technique, p.24.
95/31	P. Kelb, D. Dams and R. Gerth	Practical Symbolic Model Checking of the full μ-calculus using Compositional Abstractions, p. 17.
95/32	W.M.P. van der Aalst	Handboek simulatie, p. 51.
95/33	J. Engelfriet and JJ. Vereijken	Context-Free Graph Grammars and Concatenation of Graphs, p. 35.
95/34	J. Zwanenburg	Record concatenation with intersection types, p. 46.
95/35	T. Basten and M. Voorhoeve	An algebraic semantics for hierarchical P/T Nets, p. 32.
96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The I^2C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/16	P. Hoogendijk and O. de Moor	What is a data type?, p. 29.