FORMAL DESIGN OF REAL-TIME SYSTEMS IN A PLATFORM-INDEPENDENT WAY

JOZEF HOOMAN* AND ONNO VAN ROOSMALEN†

Abstract. To design distributed real-time systems, we distinguish two activities: (1) a platform-independent programming activity and (2) the realization of the resulting program on a particular execution platform. This paper concentrates on the first activity, proposing an extension of conventional (non-real-time) programming languages with so called timing annotations that specify the required timing constraints on an abstract level. A formal, axiomatic, semantics of a simple programming language with timing annotations is formulated, using specifications that are based on pre- and postconditions. Program correctness is independent of any underlying execution platform and can be proved by means of compositional proof rules. This is illustrated by a small example of a control system.

1. Introduction. It is generally recognized that problem decomposition and stepwise refinement [1, 11] 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, specified in terms of its interface, can be implemented independently of other components. 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. dependences that are not explicitly mentioned in the interface description of 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.

Non-real-time programming approaches provide a context and platform-independent way of describing algorithms. 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.

1.1. Platform-independent real-time programming. As a solution to the previously mentioned composability problems, we distinguish two activities in real-time software development:

1. a completely platform-independent programming activity, and
2. a system generation activity where all platform and context dependencies are addressed.

During the first activity, we achieve platform independence by extending the programming language with *timing annotations* that enable the expression of timing constraints without describing the implementation. Using our timing annotations it is possible to limit the programmed timing constraints to the ones that are implied by the problem specification. No additional constraints need to be programmed to realize or "implement" the timing requirements specified in the problem. Timing annotations thus allow a minimally constrained solution to a real-time problem.

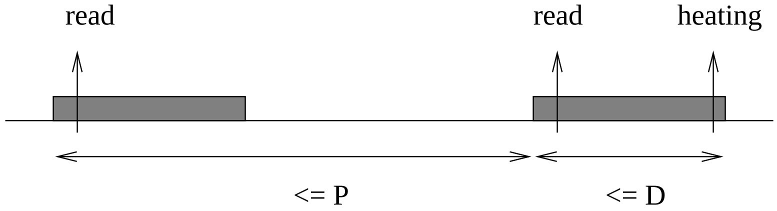
In the system generation activity it must be established that the program can be realized on the selected platform, i.e. that the platform is powerful enough to satisfy the timing constraints that are expressed in the program. This usually comes down to scheduling the application appropriately. The main difference between such a system generation activity 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). Then the platform can be changed or the program can be adapted.

*Dept. of Computing Science. Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands (hooman@win.tue.nl).

†Dept. of Computing Science. Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands (wsinonno@win.tue.nl).

Example 1 Consider, as a simple example, a heating system. If the temperature drops below a critical level, a heating unit should be started within δ time units. Suppose the temperature can be read from a certain device register.

Often such a specification is directly implemented by choosing a period P in which an execution block is scheduled. This block contains a read of the register and, if needed, activation of the heating unit. Let D be the worst-case execution time of this block.



Observe that the temperature might drop below the critical level just after the device register has been read. Then it takes P time units before the register is read again and then at most D time units before the heating is switched on. Hence $P + D \leq \delta$ is required. Note that a large reading period gives a low load of the system, but leads to a strong constraint on the worst-case execution time D. Clearly the choice of the values of P and D depends on the execution platform. Moreover, frequently hard-coded priorities are used to schedule the execution blocks of several processes appropriately, thus introducing a strong context dependency.

During the first activity of our approach we design untimed programs for the processes. For instance, for the heating system this programs is a simple loop in which the register is read and, if needed, the heating unit is activated. Timing is only specified by means of timing annotations, expressing that the heating action should occur within δ after the previous read.

In the second activity, we derive scheduling blocks and timed precedence relations from the annotated program. Given a particular execution platform, the worst-case execution times can be computed and a feasible schedule has to be found. *End Example*

1.2. Compositional Verification. It is important to note that in our approach 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. This makes it possible to apply compositional verification techniques to our programs with timing annotations. A formal method is called *compositional* if it allows reasoning with the specifications of components, without knowing their implementation.

In this paper we formulate a compositional proof method, that is, a set of axioms and compositional rules that can be used to establish program correctness against the functional as well as the timing specification, independent of the context and the platform. It can also be considered an axiomatic semantics of the programming language, including a formalization of timing annotations. Our specifications are based on Hoare triples [3] (precondition, program, postcondition) that have been extended to deal with real-time.

Not treated here is the verification involved in the second activity; establishing the existence of a feasible and correct schedule (i.e. one that satisfies the constraints expressed in the program) to execute the system on a specified platform.

1.3. Overview. Summarizing, we propose a real-time programming approach with the following properties:

- timing annotations and a programming notation for the functional behaviour are integrated in a single 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 in a compositional way (i.e. program component can be proved correct taking into account relevant interfaces of other components only, not their implementation).

The remainder of this paper is structured as follows. Section 2 contains a simple programming language and we explain how timing aspects of systems can be expressed in a platform-independent way. In section 3 we give an axiomatic semantics of this language. This also provides a method to prove correctness, as illustrated by an example of a small control program of a water level monitoring system. Concluding remarks can be found in section 4.

2. A simple parallel programming language. In previous work, our approach has been introduced informally in the object-oriented programming language *Deal*, as part of the Dependable Distributed Operating System (Dedos) project [2]. Here we concentrate on the formal semantics and verification of platform-independent real-time programs. For this purpose we introduce a simple programming language and show how it can be augmented with timing annotations. This also indicates that the basic ideas are more generally applicable to any programming notation describing functional behaviour.

Here our starting point is a normal imperative programming language. Timing constraints are expressed by adding annotations to statements. To illustrate the basic idea, consider the simple example

$$in(d_1, x)[\, m?\,]\,;\ \ldots\ \ y := f(x)\ \ \ldots\,;\ out(d_2, y)[\ < m + U\,]$$

The functional part of the program,

$$in(d_1, x)\,;\ \ldots\ \ y := f(x)\ \ \ldots\,;\ out(d_2, y)$$

expresses that the value of device register d_1 is stored in x, next some computations are done, and then device register d_2 gets the value of y. The timing annotations between the brackets $[\,\cdots\,]$ express that first the execution moment of $in(d_1, x)$ is assigned to timing variable m. (Typically the execution moment of $in(d_1, x)$ is the moment the value x is taken from the register.) Then the timing annotation of the $out(d_2, y)$ statement expresses that the execution moment of this statement falls before $m + U$.

To give a formal description of the syntax, let *VAR* be a nonempty set of program variables, with typical elements $x, x_1, \ldots, x_n, y, \ldots$ and *RTVAR* be a nonempty set of timing variables, such as $m, m_0, m_1, \ldots, m_n, \ldots$. We also introduce a nonempty set of device variables *DEV*, with typical elements $d, d_1, \ldots, d_n, \ldots$, which enable communication with devices. It is split into devices that are read, *RDEV*, and devices that are written *WDEV*. Let *CONST* be a domain of constants such as $0, 1, \ldots$, using μ for an element of this set. These sets are disjunct, e.g., $VAR \cap RTVAR = \emptyset$. The syntax of our programming language is given in the next table.

Value Expression	$e ::=$	$\mu \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::=$	$true \mid e_1 = e_2 \mid e_1 < e_2 \mid$ **not** $b \mid b_1$ **or** $b_2 \mid b_1$ **and** b_2
Moment Expression	$me ::=$	$\mu \mid m \mid me_1 + me_2 \mid me_1 - me_2 \mid me_1 \times me_2$
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	$S ::=$	$PS \mid PS[\,TA\,] \mid S_1 \,;\, S_2 \mid$ **if** b **then** S_1 **else** S_2 **fi** $\mid$ **while** b **do** S **od**
Program	$P ::=$	$S \mid P_1 \parallel P_2$

We first describe the informal meaning of the normal programming constructs and then explain the added timing annotations.

Informal meaning conventional programming language

Most of the programming constructs are well-known and are assumed to have the conventional untimed semantics in which no assumptions are made about the speed of actions. Statement $in(d, x)$ denotes reading a device variable (register) d and assigning the obtained value to the variable x.

Statement $out(d, e)$ denotes writing the value of expression e into the device register d. For the parallel composition $P_1 \parallel P_2$ we require that the variables of P_1 and P_2 are disjoint (i.e., there are no global variables). For simplicity, inter-process communication has not been included in the syntax here. The basic primitives for communication via channels are discussed in section 4.
We use **if** b **then** S **fi** as an abbreviation of **if** b **then** S **else** **skip** **fi**.

Informal meaning timing annotations

New is the addition of a time-annotated primitive statement, $PS[\,TA\,]$. 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 activity, 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. 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.

To describe the meaning of $PS[\,TA\,]$ 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, using $\equiv$ to denote syntactic equality.

- 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. E.g., **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.

Henceforth $[\,TA\,]$ is used as an abbreviation of **skip**$[\,TA\,]$.
Example 2 Consider the following annotated program.

$[\,m := 0\,]\,;$
while $true$ **do**
    $in(d_1, x)[\,>m+L\,;\,<m+U\,;\,?m\,]\,;$
    $\ldots$
    $out(d_2, y)[\,<m+\delta\,]$
od

This expresses that a read of d_1 occurs between L and U after the previous one (or 0 if there is no such one). Further, it is specified that a write of d_2 occurs in less than δ time units after the preceding read. *End Example*

Given a particular program, all execution traces are allowed that satisfy the specified timing constraints and additionally guarantee progress (any enabled programming statement will eventually be executed) and finite variability (only a finite amount of progress will happen within a finite amount of time). Additionally, reading and writing of device variables takes a positive amount of time. Apart from this nothing is known about the speed of statement execution.

To solve a real-time programming problem, it is usually not necessary for the programmer to put constraints on the execution moment 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 every statement in between: this can be deferred to the system generation activity.

Observe that expressions in the annotations are syntactically isolated from the rest of the program by using square brackets. We distinguish time domain variables (variables of $RTVAR$ used in between square brackets) and program variables (of VAR). Since $RTVAR$ and VAR are disjoint, the scope of variables is restricted to one domain only. This way separability of real-time constraints and functional aspects is achieved. In principle this syntactic restriction is not necessary, but it has been introduced for the following reasons.

1. By not introducing timing variables in the program domain, it is possible to first construct a functionally correct program, and then consider timing requirements.
2. By forbidding the use of program variables in the time domain, it becomes syntactically impossible to introduce data dependencies in the timing requirements. Such data-dependencies usually complicate correctness proofs considerably.

3. Formal verification of programs with timing annotations. A formal, axiomatic, semantics of the language is presented by formulating rules and axioms 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 [5]. Next the axiomatic semantics is given by formulating a set of compositional rules and axioms. This is also called a proof system, since it can be used to prove program correctness. 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.

3.1. Specifications. Specifications have the form $\langle\!\langle A\rangle\!\rangle\ S\ \langle\!\langle C\rangle\!\rangle$ 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 last *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.

Example 3 We present a few simple examples of specifications. The first example is similar to classical Hoare triples, except that we additionally express that the execution moment of a statement (as denoted in the commitment) is not smaller than that of the preceding statement.

$\langle\!\langle x = 3 \wedge em = 5 \wedge term\rangle\!\rangle\ \ x := x + 1\ \ \langle\!\langle x = 4 \wedge em \geq 5 \wedge term\rangle\!\rangle$.

We can generalize specifications by using logical (rigid) variables, such as v and t below.

$\langle\!\langle x = v \wedge em = t \wedge term\rangle\!\rangle\ \ x := x + 1\ \ \langle\!\langle x = v + 1 \wedge em \geq t \wedge term\rangle\!\rangle$.

The next two examples show some consequences of timing annotations.

$\langle\!\langle x = 3 \wedge em = 5 \wedge term\rangle\!\rangle\ \ x := x + 1[\,?m\,]\ \ \langle\!\langle x = 4 \wedge m \geq 5 \wedge em \geq 5 \wedge term\rangle\!\rangle$.

$\langle\!\langle x = 3 \wedge term\rangle\!\rangle\ \ x := x + 1[\ > 6\ ;\ < 9\,]\ \ \langle\!\langle x = 4 \wedge 6 < m < 9 \wedge term\rangle\!\rangle$. *End Example*

For a predicate P **at** t, expressing that P occurs at time t, and a set (usually an interval) $I \subseteq TIME$, we use the abbreviations

$$P \textbf{ during } I \equiv \forall t \in I : P \textbf{ at } t$$
$$P \textbf{ in } I \equiv \exists t \in I : P \textbf{ at } t$$

3.1.1. Specification of device registers. The programming language offers device registers for the program to communicate with its environment. A device register can either be set by the environment and read by one process in the program or it can be set by one of the processes and read by the environment. In the former case the state of the environment will be reflected in the value obtained by the read action, in the latter the program will determine the state of the device and influence the environment that way. 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 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 $d(t) = v$, and similarly for $(d > v)$ **at** t, etc.

The next property expresses that the value read corresponds to the actual value of the device register.

PROPERTY 1 (ReadVal). *For $d \in RDEV$,*

$$read(d, v) \textbf{ at } t \to d(t) = v$$

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

PROPERTY 2 (WriteVal). *For $d \in WDEV$,*

$$\begin{aligned} d(t) = v \leftrightarrow & (\exists t_0 \leq t : set(d, v) \textbf{ at } t_0 \wedge \forall v_0 \neq v : \neg set(d, v_0) \textbf{ during } [t_0, t]) \vee \\ & (v = d(0) \wedge \neg set(d) \textbf{ during } [0, t]) \end{aligned}$$

At most one value is written at any point in time.

PROPERTY 3 (Uniqueness). *For $d \in WDEV$,*

$$set(d, v_1) \textbf{ at } t \wedge set(d, v_2) \textbf{ at } t \to v_1 = v_2$$

Finally, only a finite number of write actions can be performed in a finite amount of time. Hence there exists a last write in any finite period I which contains at least one write.

PROPERTY 4 (Finiteness). *For $d \in WDEV$,*

$$set(d) \textbf{ in } I \to \exists t \in I : set(d) \textbf{ at } t \wedge (\neg set(d)) \textbf{ during } \{t_0 \in I \mid t_0 > t\}$$

Example 4 We present a few simple specifications, to give an indication of what can be expressed. First consider a statement $in(d, y)$ which reads a device (taking a positive amount of time).

$\langle\langle x = 7 \wedge em = 3 \wedge term \rangle\rangle$
$\quad in(d, y)$
$\langle\langle x = 7 \wedge em > 3 \wedge read(d, y) \textbf{ at } em \wedge term \rangle\rangle$.

The next example shows an *out* statement and the use of logical variables (v and t).

$\langle\langle x = v \wedge em = t \wedge read(d_1, 6) \textbf{ at } 5 \wedge term \rangle\rangle$
$\quad out(d_2, x + 3)$
$\langle\langle x = v \wedge em > t \wedge read(d_1, 6) \textbf{ at } 5 \wedge set(d_2, v + 3) \textbf{ at } em \wedge term \rangle\rangle$.

The following example contains timing annotations.

$\langle\langle x = 1 \wedge \delta = 10 \wedge m = 15 \wedge term \rangle\rangle$
$\quad out(d, x + 3)[\ < m + \delta\ ]$
$\langle\langle x = 1 \wedge set(d, 4) \textbf{ at } em \wedge em < 25 \wedge term \rangle\rangle$.

The next, non-terminating, program sets d infinitely often with a distance of at most δ time units.
$\langle\langle em = 0 \wedge term \rangle\rangle$

$\quad [\, m := 0 \,]\,;$

$\quad$ **while** $true$ **do** $out(d, 0)[\, < m + \delta \,;\, m? \,]$ **od**

$\langle\langle \neg term \wedge \forall t : set(d, 0)$ **in** $[t, t+\delta] \rangle\rangle$. *End Example*

3.2. Proof system. We briefly present the rules and axioms for our programming language and refer to [5] for more explanation on the rules for the conventional programming constructs. We assume that all new (logical) variables which are introduced in the rules are fresh.

3.2.1. Axiomatization of Programming Constructs. Let $A[exp/x]$ denote the assertion obtained by substituting expression exp for all free occurrences of variable x in A. In the rule for a **skip** statement, the last execution moment before this statement is represented by t_0 (replacing em in A by t_0).

RULE 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 $em = t_0$). In the rule for assignment $x := e$, similarly the execution moment of the preceding statement is represented by t_0 and the initial value of x is represented by v (by means of the substitution $A[t_0/em, v/x]$).

RULE 2 (Assignment).

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

Example 5 The formula
$\langle\langle x = 4 \wedge 5 < em \wedge em < 7 \wedge term \rangle\rangle \ \ x := x + 2 \ \ \langle\langle x = 6 \wedge em > 5 \wedge term \rangle\rangle$
can be derived by the assignment rule, since
$(x = 4 \wedge 5 < em \wedge em < 7)[t_0/em, v/x] \wedge$ $term \wedge x = (x+2)[v/x] \wedge em \geq t_0$ equals
$v = 4 \wedge 5 < t_0 \wedge t_0 < 7 \wedge$ $term \wedge x = v + 2 \wedge em \geq t_0$ which implies
$x = 6 \wedge em > 5 \wedge term$. *End Example*

Similarly, rules for access to device registers are formulated.

RULE 3 (Read).

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

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

RULE 4 (Set).

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

The rules for sequential composition and choice are identical to the classical rules of Hoare logic [3].

RULE 5 (Sequential Composition).

$$\frac{\langle\langle A \rangle\rangle \ S_1 \ \langle\langle B \rangle\rangle, \quad \langle\langle B \rangle\rangle \ S_2 \ \langle\langle C \rangle\rangle}{\langle\langle A \rangle\rangle \ S_1 ; S_2 \ \langle\langle C \rangle\rangle}$$

Rule 6 (If-then-else).

$$\frac{\langle\!\langle A \wedge b \rangle\!\rangle\ S_1\ \langle\!\langle C \rangle\!\rangle, \quad \langle\!\langle A \wedge \neg b \rangle\!\rangle\ S_2\ \langle\!\langle C \rangle\!\rangle}{\langle\!\langle A \rangle\!\rangle\ \mathbf{if}\ \ b\ \ \mathbf{then}\ \ S_1\ \ \mathbf{else}\ \ S_2\ \ \mathbf{fi}\ \langle\!\langle C \rangle\!\rangle}$$

The rule for the while construct is more complicated, since our framework considers both terminating and non-terminating computations. For simplicity, we only present the rule for a while construct with a true boolean guard. Clearly **while** *true* **do** S **od** does not terminate and there are two classes of computations: either S always terminates and we have an invariant I which holds after each execution of S for increasing execution moments (represented by I_1 below) or eventually S does not terminate (represented by its invariant I).

Rule 7 (While true).

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

See [5] for a general rule, and similarly for more details on sequential and parallel composition.

Rule 8 (Parallel Composition).

$$\frac{\langle\!\langle A_1 \rangle\!\rangle\ P_1\ \langle\!\langle C_1 \rangle\!\rangle, \qquad \langle\!\langle A_2 \rangle\!\rangle\ P_2\ \langle\!\langle C_2 \rangle\!\rangle}{\langle\!\langle A_1 \wedge A_2 \rangle\!\rangle\ P_1 \| P_2\ \langle\!\langle C_1 \wedge C_2 \rangle\!\rangle}$$

provided

- the commitment of one process should not refer to local objects of the other process;
- assertions in the specification of a process refer only to the interface of the process itself;
- em and $term$ do not occur in the commitments C_1 and C_2.

3.2.2. Axiomatization of Timing Annotations. Next we introduce a few new 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 $\langle\!\langle\!\langle A \rangle\!\rangle\!\rangle\ [\,TA\,]\ \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle$.

As a simple example, to show the main idea, consider the formula

$\langle\!\langle x = 3 \wedge m_0 = 5 \wedge term \rangle\!\rangle$
$\quad out(d, x+3)[\ > m_0 + 2\ ;\ < m + 4\ ;\ ?m_0\ ;\ m_0 := m_0 + 10\,]$
$\langle\!\langle x = 3 \wedge set(d,6)\ \mathbf{at}\ em \wedge em > 7 \wedge\ em < m + 4 \wedge m_0 = em + 10 \wedge term \rangle\!\rangle$.

This is proved by first deriving

$\langle\!\langle x = 3 \wedge m_0 = 5 \wedge term \rangle\!\rangle$
$\quad out(d, x+3)$
$\langle\!\langle x = 3 \wedge m_0 = 5 \wedge set(d,6)\ \mathbf{at}\ em \wedge term \rangle\!\rangle$

with the Set Rule above. Next we will present rules to derive

$\langle\!\langle\!\langle x = 3 \wedge m_0 = 5 \wedge set(d,6)\ \mathbf{at}\ em \wedge term \rangle\!\rangle\!\rangle$
$\quad [\ > m_0 + 2\ ;\ < m + 4\ ;\ ?m_0\ ;\ m_0 := m_0 + 10\,]$
$\langle\!\langle\!\langle x = 3 \wedge set(d,6)\ \mathbf{at}\ em \wedge em > 7 \wedge\ em < m + 4 \wedge m_0 = em + 10 \wedge term \rangle\!\rangle\!\rangle$.

Then the following rule leads to the desired formula.

Rule 9 (Timing Annotation Introduction).

$$\frac{\langle\!\langle A \rangle\!\rangle\ S\ \langle\!\langle B \rangle\!\rangle, \quad \langle\!\langle\!\langle B \rangle\!\rangle\!\rangle\ [\,TA\,]\ \langle\!\langle\!\langle C \rangle\!\rangle\!\rangle}{\langle\!\langle A \rangle\!\rangle\ S[\,TA\,]\ \langle\!\langle C \rangle\!\rangle}$$

To derive formulas of the form $\langle\langle\langle A\rangle\rangle\rangle\ [\,TA\,]\ \langle\langle\langle C\rangle\rangle\rangle$ we introduce three axioms and a rule.

Axiom 1 (Timing Assignment).

$$\langle\langle\langle C[me/m]\rangle\rangle\rangle\ [\,m := me\,]\ \langle\langle\langle C\rangle\rangle\rangle$$

Axiom 2 (Time Measurement).

$$\langle\langle\langle C[em/m]\rangle\rangle\rangle\ [\,?m\,]\ \langle\langle\langle C\rangle\rangle\rangle$$

Example 6 As a simple example, observe that we can derive $\langle\langle\langle em = 5\rangle\rangle\rangle\ [\,?m\,]\ \langle\langle\langle m = 5\rangle\rangle\rangle$. *End Example*

Axiom 3 (Timing Requirement).

$$\langle\langle\langle A\rangle\rangle\rangle\ [\,R\,me\,]\ \langle\langle\langle A \wedge em\ R\ me\rangle\rangle\rangle$$

Rule 10 (Timing Annotation Sequential Composition).

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

Example 7 With these rules we can derive, e.g., 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**
$\langle\langle m_2 \leq m_1 \leq m_2 + max(\delta_1, \delta_2)\rangle\rangle$. *End Example*

3.2.3. General Rules and Axioms. The proof system contains several general rules such as the consequence rule which allows strengthening of assumptions and weakening of commitments.

Rule 11 (Consequence).

$$\frac{\langle\langle A_0\rangle\rangle\ P\ \langle\langle C_0\rangle\rangle, A \to A_0, C_0 \to C}{\langle\langle A\rangle\rangle\ P\ \langle\langle C\rangle\rangle}$$

Here we only list the rules pertaining to the particulars of our real-time language, and refer to [5] for other rules. The first axiom expresses that a terminating program takes only a finite amount of time.

Axiom 4 (Finite Time).

$$\langle\langle em = t_0 \wedge term\rangle\rangle\ S\ \langle\langle term \to \exists\delta \geq 0 : em \leq t_0 + \delta\rangle\rangle$$

For a statement S, let $write(S)$ be the set of device 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 device variables that might be read by S, i.e. the set of variables d such that $in(d, x)$ occurs in S. We assume that at most one process can access a device variable.

Axiom 5 (Read Invariance). If $d \notin read(S)$,

$$\langle\langle em = t_0\rangle\rangle\ S\ \langle\langle(\neg read(d))\ \textbf{during}\ (t_0, em]\rangle\rangle$$

Axiom 6 (Write Invariance). If $d \notin write(S)$,

$$\langle\langle em = t_0\rangle\rangle\ S\ \langle\langle(\neg set(d))\ \textbf{during}\ (t_0, em]\rangle\rangle$$

3.3. Verification of a simple control system. We specify, implement and verify the control program of a water level monitoring 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. Using the maximal flow in and out of the vessel and the critical levels, the response time D for the control system to switch the pump on or off is calculated. This leads to the control requirement

$\forall t_1, t_2 :$ (water critically high) **during** $[t_1, t_2] \rightarrow$ (pump is on) **during** $[t_1 + D, t_2]$

and similarly for the critically low value. Because of symmetry we only show the high value case here. For simplicity we assume that two device variables, water and pump, represent the water level and the pump state instantaneously. Assume pump ranges over $\{on, off\}$. The aim is to design a control system satisfying

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

Since the right-hand side is trivially true if $t_1 + D > t_2$, it is equivalent to

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

Similarly we have CTL_2 for $\mathsf{water} < \mathsf{low}$ and $\mathsf{pump} = off$. The aim is to design a control system Contr such that

$\langle\!\langle em = 0 \rangle\!\rangle$ Contr $\langle\!\langle CTL_1 \wedge CTL_2 \rangle\!\rangle$.

The program must keep the pump in the correct state. It can do so by setting the variable pump at selected moments. Hence we refine the specification to

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

Similarly, CP_2 is defined for $\mathsf{water} < \mathsf{low}$. Then we can prove the following lemma.

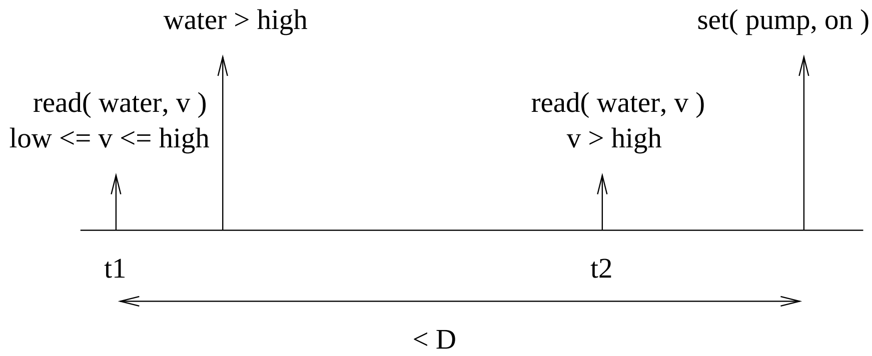
LEMMA 3.1. $CP_1 \wedge CP_2$ implies $CTL_1 \wedge CTL_2$

Proof. To prove CTL_1, assume for t_1, t_2 that $t_1 + D \leq t_2$ and $(\mathsf{water} > \mathsf{high})$ **during** $[t_1, t_2]$. By CP_1, there exists a t_3 with $t_3 \leq t_1 + D$, $set(\mathsf{pump}, on)$ **at** t_3, and $(\neg set(\mathsf{pump}, off))$ **during** $[t_3, t_2]$. By Property 2 (WriteVal) this implies $(\mathsf{pump} = on)$ **during** $[t_3, t_2]$, and hence $(\mathsf{pump} = on)$ **during** $[t_1 + D, t_2]$.
The proof of CTL_2 proceeds similarly. □

Hence it remains to implement Contr such that

$\langle\!\langle em = 0 \rangle\!\rangle$ Contr $\langle\!\langle CP_1 \wedge CP_2 \rangle\!\rangle$.

Next we use device variable water to read the value of the water level. A possible step in the design is now to choose a certain reading period R and to specify that the pump should be switched on in less than $D - R$ time units when reading a value greater than high. Here we avoid the choice of periods and corresponding internal reaction times, by specifying that a response is needed in less than D time units after the previous read, since this represents the worst-case assumption about the water level. See the figure below.



To specify this design idea, we do not explicitly refer to the previous read, but only to the preceding interval without read actions. Abbreviation $ReadPeriod(t_1, t_2, v)$ expresses that after t_1 the first read action of water is at t_2 and value v is read.

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

Next a few additional abbreviations.

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

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

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

To specify the control program, commitment CC_1 expresses that a value is read at least once every period of D time units.

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

If the first read after t_1 takes place at t_2 and yields a high value then the pump should be switched on. Since it is possible that the water was already high immediately after t_1, the deadline on the set action is $t_1 + D$.

$CC_2 \equiv \forall t_1, t_2 : ReadPeriodHigh(t_1, t_2) \rightarrow set(\mathsf{pump}, on)$ **in** $[t_2, t_1 + D]$

Finally we specify that the pump is only switched off at time t if a low value has been read at t_2 before t and no value has been read between t_2 and t.

$CC_3 \equiv \forall t : set(\mathsf{pump}, off)$ **at** $t \rightarrow \exists t_1, t_2 : t \in [t_2, t_1 + D] \wedge ReadPeriodLow(t_1, t_2) \wedge$
$(\neg read(\mathsf{water}))$ **during** $(t_2, t]$

In addition, the symmetric case is described by CC_4 and CC_5, interchanging high and low, and also *on* and *off*. Let $CC \equiv CC_1 \wedge CC_2 \wedge CC_3 \wedge CC_4 \wedge CC_5$. Then we have the following lemma.

LEMMA 3.2. If $\mathsf{low} \leq \mathsf{high}$ then CC implies $CP_1 \wedge CP_2$.

Proof. By symmetry, it is sufficient to show CP_1. Suppose the premise of the implication in CP_1 is satisfied and there exists a t_1 and t_2 with

$$t_1 + d \leq t_2 \wedge (\mathsf{water} > \mathsf{high}) \textbf{ during } [t_1, t_2] \tag{3.1}$$

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

$$\begin{aligned} &t_{11} < t_1 \leq t_{21} \leq t_1 + D \wedge read(\mathsf{water}, v) \textbf{ at } t_{21} \wedge \\ &\qquad (\neg read(\mathsf{water})) \textbf{ during } (t_{11}, t_{21}) \end{aligned} \tag{3.2}$$

Observe that $t_{21} \in [t_1, t_2]$, thus assumption (3.1) leads to $\mathsf{water}(t_{21}) > \mathsf{high}$. By Property 1 (ReadVal) we have $\forall t : read(\mathsf{water}, v)$ **at** $t \rightarrow \mathsf{water}(t) = v$. Hence $v > \mathsf{high}$, and thus

$$ReadPeriodHigh(t_{11}, t_{21}) \tag{3.3}$$

By CC_2 we obtain $set(\mathsf{pump}, on)$ **in** $[t_{21}, t_{11} + D]$. That is, there exists a t_3 with

$$t_{21} \leq t_3 \leq t_{11} + D \wedge set(\mathsf{pump}, on) \textbf{ at } t_3 \tag{3.4}$$

Since, using (3.2), $t_3 \leq t_{11} + D < t_1 + D$, it remains to show $(\neg set(\mathsf{pump}, off))$ **during** $[t_3, t_2]$.

Suppose $set(\mathsf{pump}, off)$ **in** $[t_3, t_2]$, i.e., assume there exists a $t_4 \in [t_3, t_2]$ with $set(\mathsf{pump}, off)$ **at** t_4. Then by CC_3 there exist t_{12} and t_{22} such that $t_4 \in [t_{22}, t_{12} + D]$, $ReadPeriodLow(t_{12}, t_{22})$, and $(\neg read(\mathsf{water}))$ **during** $(t_{22}, t_4]$. Given the last formula, observing that $t_4 \geq t_3 \geq t_{21}$ and using $read(\mathsf{water})$ **at** t_{21} from (3.2), we obtain that $t_{21} \leq t_{22}$. By (3.2) we have $t_1 \leq t_{21}$ thus $t_1 \leq t_{22}$. Since $t_{22} \leq t_4 \leq t_2$ this leads to $t_{22} \in [t_1, t_2]$. Then (3.1) implies that $\mathsf{water}(t_{22}) > \mathsf{high}$, and thus $\mathsf{low} < \mathsf{high}$ leads to $\neg(\mathsf{water}(t_{22}) < \mathsf{low})$. By Property 1 (ReadVal) this yields a contradiction with $ReadPeriodLow(t_{12}, t_{22})$.

Hence $(\neg set(\mathsf{pump}, \mathit{off}))$ **during** $[t_3, t_2]$ and CP_1 has been proved. □

Consequently, control program Contr is specified by

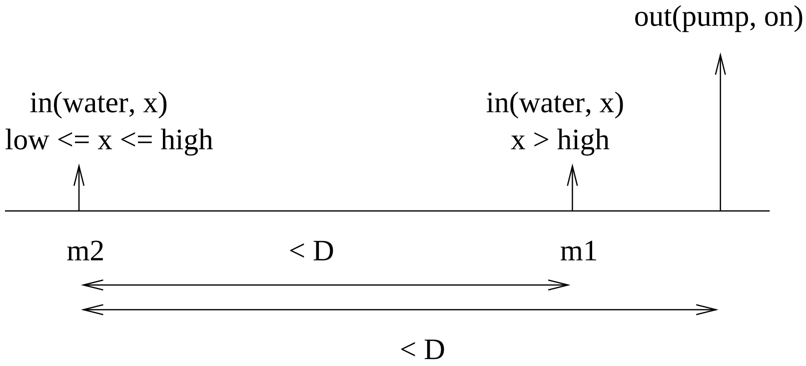
$\langle\!\langle em = 0 \rangle\!\rangle$ Contr $\langle\!\langle CC \rangle\!\rangle$.

Note that CC_3 is stronger than strictly necessary for the proof. It incorporates the design decision that no additional read action will take place between reading a low water level and setting the pump. Similarly for CC_5. This excludes programs in which such read and set actions are performed by different processes and they need not alternate.

To satisfy the commitment CC we propose the following program.

Contr $\equiv$
$[\, m_1 := 0\,]$;
while *true* **do**
  $[\, m_2 := m_1\,]$;
  $in(\mathsf{water}, x)[\ < m_2 + D\ ;\ ?m_1\,]$;
  if high $< x$ **then** $out(\mathsf{pump}, on)[\ < m_2 + D\,]$
  else **if** $x <$ low **then** $out(\mathsf{pump}, \mathit{off})[\ < m_2 + D\,]$
  fi fi
od

The program should be read as follows. Initially timing variable m_1 is set to 0. On the first sweep of the repetition, m_2 obtains this value in the first annotated statement. The first execution of the *in* statement takes place before time D, and m_1 records the actual execution moment. If subsequently a pump action is required, the state change corresponding to the relevant *out* statement is realized before the deadline D. In the next sweep m_2 gets the old value of m_1 and m_1 records the execution moment of the next *in* statement that inspects the environment. A possible response to the value obtained must now take place before $m_2 + D$, i.e. within time D from the one but last inspection. Since the *out* statements need not be executed, the maximal distance D between successive reads is specified separately. This pattern is repeated forever.



Considering the specification, this program can be motivated as follows. CC_1 is satisfied by allowing at most D time units to pass between two consecutive sensor readings. This is expressed in the annotation of the *in* statement. The maximum *ReadPeriod* that ends at a particular read action extends backwards to the moment of the previous read action. 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 inspection of the water level. Similarly, it is easy to see that CC_3 holds, since the pump is only switched off within D time units after a low value has been read.

LEMMA 3.3. $\langle\!\langle em = 0 \rangle\!\rangle$ Contr $\langle\!\langle CC \rangle\!\rangle$

Proof. To prove CC_1, 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 proof is based on a loop invariant I_1 which expresses essentially that $CC_1(t)$ holds for all $t < m_1$, where m_1 is a timing variable. Since additionally $m_1 \geq em - D$, the property that the while loop yields unbounded values of em leads to $CC_1(t)$, for all t.

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

The proof is represented by a so called proof outline; a program where assertions have been inserted at the location where they should hold. 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 $\langle\!\langle A \rangle\!\rangle\ S\ \langle\!\langle B \rangle\!\rangle\ TA\ \langle\!\langle C \rangle\!\rangle$. In the proof outline below we use the auxiliary assertion A_1.

$A_1 \equiv term \wedge m_2 < m_1 < em \wedge \neg read(\mathsf{water})$ **during** $(m_2, em) \wedge read(\mathsf{water})$ **at** $m_1 \wedge$
$\quad \forall t < m_2 : CC_1(t)$

We assume that initially $em = 0$, representing the assumption that nothing is executed before this program.

$\langle\!\langle term \wedge em = 0 \rangle\!\rangle$
$[\, m_1 := 0\,]\,;\ \langle\!\langle term \wedge m_1 = em = 0 \rangle\!\rangle$
while $true$ **do** $\langle\!\langle I_1 \rangle\!\rangle$
$\quad [\, m_2 := m_1\,]\,;\ \langle\!\langle B_1 \equiv I_1[m_2/m_1] \rangle\!\rangle$
$\quad in(\mathsf{water}, x)\ \langle\!\langle C_1 \equiv B_1 \wedge read(\mathsf{water})\ \textbf{at}\ em \rangle\!\rangle\ [\ < m_2 + D\ ;\ ?m_1\ ]\,;$
$\quad \langle\!\langle D_1 \equiv C_1 \wedge em < m_2 + D \wedge m_1 = em \rangle\!\rangle$
$\quad$ **if** $\mathsf{high} < x$ **then** $\langle\!\langle D_1 \rangle\!\rangle\ out(\mathsf{pump}, on)\ \langle\!\langle A_1 \rangle\!\rangle\ [\ < m_2 + D\ ]\ \langle\!\langle I_1 \rangle\!\rangle$
$\quad$ **else if** $x < \mathsf{low}$ **then** $\langle\!\langle D_1 \rangle\!\rangle\ out(\mathsf{pump}, off)\ \langle\!\langle A_1 \rangle\!\rangle\ [\ < m_2 + D\ ]\ \langle\!\langle I_1 \rangle\!\rangle$
$\quad$ **fi fi** $\langle\!\langle I_1 \rangle\!\rangle$
od
$\langle\!\langle CC_1 \rangle\!\rangle$

The last step, the proof of CC_1, follows from Rule 7 (While true), 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.

To prove CC_2 define

$CC_2(t_2) \equiv \forall t_1 : ReadPeriodHigh(t_1, t_2) \rightarrow set(\mathsf{pump}, on)$ **in** $[t_2, t_1 + D]$

Then proof proceeds similarly to the one above, using the invariant

$I_2 \equiv term \wedge \neg read(\mathsf{water})$ **during** $(m_1, em) \wedge (m_1 = 0 \vee read(\mathsf{water})$ **at** $m_1) \wedge \forall t < em : CC_2(t)$

Similarly, CC_3 is proved by means of

$CC_3(t) \equiv set(\mathsf{pump}, off)$ **at** $t \rightarrow \exists t_1, t_2 : t \in [t_2, t_1 + D] \wedge ReadPeriodLow(t_1, t_2) \wedge$
$\qquad (\neg read(\mathsf{water}))$ **during** $(t_2, t]$

and the invariant

$I_3 \equiv term \wedge \neg read(\mathsf{water})$ **during** $(m_1, em) \wedge \forall t < em : CC_3(t)$ □

4. Concluding remarks. We have described how a programming language can be extended with timing annotations to describe real-time constraints. The syntax of the programming language used here was just a vehicle to illustrate the approach and details of the syntax and semantics are not essential. We believe, however, that the following properties *are* essential for any successful approach to real-time programming.

1. Formal techniques are essential, at least to define a real-time programming paradigm. Many current proposals are hard to understand and difficult to apply because a formal semantics of new constructs is not provided.
2. A strict separation of concerns in the programming and system generation activities should be used. All platform and context dependencies should be addressed during system generation. In this respect, the only difference with non-real-time programming is that the system generator may decide not to implement the program on the specified platform.

3. Real-time programs *can* and *should* be designed *and verified* independently of an execution platform.
4. Over-specification of timing behavior should be avoided. Usually over-specifications are made on the basis of platform considerations.

An important aspect of our work is the possibility to formally establish program correctness without making any assumptions about the platform that will execute the program.

Concerning the second point, observe that in contrast with other formal methods for real-time systems (see, e.g., [8]), in our approach program correctness does not require any assumption about the underlying platform. No assumptions are made about the duration of statements, the number of processors, the mapping of processes to processors, the scheduling policy, etcetera. We are not aware of any other programming framework that provides a similar rigorous abstraction from the execution platform.

4.1. Communication between parallel processes. In a technical report [6] we have given a more detailed description of our approach, illustrated by a larger example of a mine pump system. In that example the design leads to a program with several parallel processes and we show that this program can be scheduled on various platforms.

To denote synchronization and communication between parallel processes in an abstract way, two new statements are introduced:

- $send[\,!me\,](c, e)$ 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 physically sent). We assume that messages are stored in an infinite FIFO buffer.
- $receive[\,?m\,](c, x)$ 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.

With this notation, there is no need to split global deadlines into local deadlines for the processes, thus avoiding over-specification (see point 4. above). The basic idea is that constraints on actions can be passed from one process to another (conceptually).

Example 8 Assume given two functions f_1 and f_2 on the data domain. It is specified that after reading a value v from register d_1, a computed value $f_2(f_1(v))$ should be written into register d_2 in less than δ time units. Thus

$\forall t, v : read(d_1, v)$ **at** $t \rightarrow set(d_2, f_2(f_1(v)))$ **in** $[t, t + \delta)$

Additional constraints, such as a certain frequency of reading input, are omitted here. Suppose we decide to implement this by means of two processes, where the first one computes f_1 and sends the result to the second process which then computes f_2. Then we can avoid the decision to split up δ into deadlines for each of the processes, which would depend on the execution time of the computations needed for f_1 and f_2. The next parallel program specifies only the constraints on the external input-output behaviour.

while *true* **do**	$\|$	**while** *true* **do**
$\quad in(d_1, x)[\,?m_1\,]$;		$\quad receive[\,?m_2\,](c, w)$;
$\quad y := f_1(x)$;		$\quad z := f_2(w)$;
$\quad send[\,!m_1 + \delta\,](c, y)$		$\quad out(d_2, z)[\,< m_2\,])$
od		**od**

Note that the annotation of the send statement does not represent a constraint, but merely propagates a constraint to the receiver. *End Example*

4.2. Comparison. Technical report [6] also contains a comparison with other real-time languages, notably Ada95 and a couple of recent object-oriented real-time languages (e.g. those published in [7, 9, 10]). We summarize here the major differences with other approaches.

Usually timing requirements pertain to the control of observable events. In our approach the related timing constraints can be formulated without constraining unnecessarily the sub-actions carried out between the statements relevant to these events. Consider for example the water level control program. In other languages strictly periodic release-times for processes that inspect sensors must be specified. This requires additional design choices. Consider in our example, e.g., the constraint on the time between a measurement of the water level and the related response of the pump. In other approaches this interval must be divided into a fixed release-period for the process making the measurement and a deadline of the pump-response within such a period. When the period is made smaller, the system load is increased; when it is made larger, the available time to the deadline is smaller and the number of platforms that can execute the process is reduced. Thus, the programmer is enticed to take platform properties into consideration. In our approach such decisions are moved to the system generation activity and the program is kept more general.

Most languages that support timing constraints use timing constraints that are attached to program blocks. In our opinion this choice is inspired by implementation considerations (the relative ease of scheduling blocks) and not by considerations of what the proper abstraction for the programmer should be. Many languages offer process priorities in addition to constructs for timing constraints. Priorities are heuristics concerned with the realization of constraints. They introduce platform and context dependencies in a program component; the choice of the priority for a particular process depends on the scheduling mechanisms offered by the platform and also depends on priorities of other processes with which no explicit interface exists.

4.3. Future work. Concerning the formal verification of programs, we are currently working with the verification system PVS to be able to check proofs mechanically. Earlier work on the compositional verification of real-time programs in PVS [4] has to be extended to be able to reason with timing annotations.

The main ideas of our approach have been devised in the context of the object-oriented language Deal. Future work will address the extension of our formal framework towards object-orientation. More research is also needed on the development of a design methodology, that is, a method to design programs with timing annotations in a systematic way.

In addition to establishing program correctness, it must be shown that a program is correctly implemented on a particular platform. Therefore, applicability of our approach requires suitable tool support for the compilation activity in which a program is mapped onto a particular platform. Some experience has been obtained by building prototypes for a timing analyzer and tool support for the generation of a static off-line schedule.

REFERENCES

[1] E.W. Dijkstra. Notes on structured programming. *A.P.I.C. Studies in Data Processing*, 8:1–81, 1972.

[2] D. Hammer, E. Luit, O. van Roosmalen, P. van der Stok, and J. Verhoosel. Dedos: A distributed real-time environment. *IEEE Parallel & Distributed Technology, Systems & Applications*, 2(4):32–47, 1994.

[3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[4] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, Springer-Verlag, 1994.

[5] J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.

[6] J. Hooman and O. van Roosmalen. A programming-language extension for distributed real-time systems. Computing Science Report 97/02, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1997.

[7] Y. Ishikawa. Object-oriented real-time language design. *ACM SIGPLAN Notices*, 25(10):289–298, 1990.

[8] H. Langmaack, W.-P. de Roever, and J. Vytopil, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*. LNCS 863. Springer-Verlag, 1994.

[9] K. Lin, J. Liu, K. Kenny, and S. Natarajan. FLEX: A language for programming flexible real-time systems. In M. van Tilborg and G. Koob, editors, *Foundations of Real-Time Computing (Formal Specifications and Methods)*, pages 251–289. North Holland, 1991.

[10] K. Takashio and M. Tokoro. DROL: An object-oriented programming language for distributed real-time systems. *ACM SIGPLAN Notices*, 27(10):276–294, 1992.

[11] N. Wirth. Programming development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.