Formal Aspects of Computing (1994) 6A: 801–825
© 1994 BCS

Formal Aspects of Computing

Extending Hoare Logic to Real-Time

Jozef Hooman

Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands

Keywords: Formal specification; Top-down design; Compositionality; Real-time; Concurrency; Hoare logic

Abstract. Classical Hoare triples are modified to specify and design distributed real-time systems. The assertion language is extended with primitives to express the timing of observable actions. Further the interpretation of triples is adapted such that both terminating and nonterminating computations can be specified. To verify that a concurrent program, with message passing along asynchronous channels, satisfies a real-time specification, we formulate a compositional proof system for our extended Hoare logic. The use of compositionality during top-down design is illustrated by a process control example of a chemical batch processing system.

1. Introduction

The general aim of this work is the development a formal framework for the specification and verification of real-time embedded systems. Usually, this concerns distributed systems which have an intensive interaction with their environment, often called reactive systems [HaP85]. Further these systems might consist of both software controlled components and physical components with a time-continuous nature; so-called hybrid systems [GNR93]. To verify (real-time) properties of such systems, we aim at a proof system in which it is possible to consider a part of the system as a black box and use its specification only. Furthermore, the method should support top-down program design. Therefore we aim at a proof system which is *compositional*, that is, for each compound programming language construct (such as sequential composition and parallel composition) there is a rule

Correspondence and offprint requests to: Jozef Hooman, Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. Email: wsinjh@win.tue.nl

in which the specification of the construct can be deduced from specifications of its constituent parts without any further information about the internal structure of these parts. By means of a compositional proof system design steps can be verified during the process of top-down program construction.

Observing the nice compositional rules for sequential composition and iteration of classical Hoare triples (precondition, program, postcondition) [Hoa69], we have modified these triples to obtain a formalism for the specification and the verification of distributed real-time reactive systems. This has been achieved by extending the assertion language, in which the precondition and the postcondition are expressed, and modifying the interpretation of triples.

As usual, in the assertion language one can refer to program variables to express the functional behaviour of a program, that is, the relation between the values of the program variables at the start of the execution and the values of these variables at termination. To express timing, we add a special variable *now* (similar to [AbL92]) which represents in the precondition the starting time of the program whereas in the postcondition it denotes the termination time. In this way we can specify, e.g., bounds on the execution time of a program. Further the real-time communication interface can be specified by means of primitives denoting the timing of observable events. In this paper we consider communication via asynchronous channels and use primitives to express when a message is transmitted, when a process is waiting to receive, and when a message is received.

With classical Hoare triples we can only specify properties of terminating computations, i.e. partial correctness. Having introduced timing primitives in the assertion language, however, we aim at a formalism in which, besides these safety properties, also liveness properties can be expressed. This goal is based on the observation of Lamport [Lam83] that the characterization of safety as "nothing bad will happen" and liveness as "eventually something good must happen" is not appropriate for real-time properties. For instance, "termination within 10 time units" and "communication via channel c within 25 time units" are safety properties (they can be falsified after, resp., 10 and 25 time units), but they express that something must happen. Further observe that the real-time safety property "termination within 10 time units" implies the liveness property "termination". Therefore we have adapted the interpretation of triples such that the postcondition has to hold for terminating as well as for nonterminating computations. Together with the timing primitives this leads to a framework in which progress properties can be specified.

The remainder of this paper is structured as follows.

- In Section 2 we introduce the basic formalism, concentrating on the parallel composition of components.
- We illustrate the method by a hybrid system, a process control example of a chemical batch processing system. This example is presented in Section 3, based on a description in [ALF93]. We give a formal specification and verify a first design step.
- Next, in Section 4, we introduce a communication mechanism for parallel processes using asynchronous channels.
- This communication mechanism is used in Section 5, where the control system of the chemical batch processing example is refined by introducing a sensor

and an actuator. This leads to the specification of a control component which is implemented in software.

- In Section 6 we introduce a concurrent programming language, including a few basic real-time constructs inspired by Occam [Occ88] and Ada [Ada83]. Proving properties of such programs requires information about implementation details from which one usually abstracts in non-real-time models, such as the execution time of atomic statements. We formulate a compositional proof system for this language using our extended Hoare logic.
- In Section 7 the control component, as specified in Section 5, is implemented in the programming language.
- Concluding remarks can be found in Section 8.

2. Basic Framework

In this section we introduce the basic framework. First we consider only the parallel composition of processes, described in Section 2.1, without further details about the implementation of processes (which could be in hardware or in software). The semantic model to describe the behaviour of real-time processes is given in Section 2.2. The formalism to specify real-time systems is presented in Section 2.3 and a few proof rules are given in Section 2.4.

2.1. Parallel Processes

In this section we abstract from the internal implementation of processes and only assume that there is a parallel composition operator $\|$. Further we do not give details of the communication mechanism between parallel components, but only assume that certain objects (e.g. channels, variables, or physical quantities) of a process can be observed by its parallel environment. Let

- $obs(P)$ be the set of (representations of) observable objects of process P.

Thus $obs(P)$ represents the interface of P. For instance, if P communicates via channels then $obs(P)$ contains the names of these channels, and if P uses shared variables then the names of these variables are included in the set. Define $obs(P_1 \| P_2) = obs(P_1) \cup obs(P_2)$. Actions of a process that affect its interface are called observable actions and the occurrence of an observable action is called an observable event.

Process P will also have local objects (e.g. local variables), represented by

- $loc(P)$, describing local objects of P that are not observable by other parallel processes.

For $P_1 \| P_2$ we assume that $loc(P_1) \cap loc(P_2) = \emptyset$. Henceforth we assume that local variables range over a value domain VAL. In examples we assume that VAL equals the set of reals $\mathbb{R}$.

Observe that the real-time behaviour of a concurrent program depends on the number of available processors. Thus assumptions have to be made about the execution of parallel processes. In this paper the maximal parallelism assumption is used, representing the situation that each process has its own processor.

2.2. Semantic Model

In this section we introduce the semantic model which is used to describe real-time computations. We describe the timing behaviour 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 behaviour 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. Then we define the real-time semantics of programs by means of a function which assigns to a point of time a set of records, representing the observable events that are taking place at that point.

We use a time domain *TIME* which is dense, i.e., between every two points of time there exists an intermediate point. With such a time domain we can easily model events that are arbitrarily close to each other in time. Having dense time is also suitable for the description of hybrid systems which interact with an environment that has a time-continuous nature (see the example in Section 3). Here we take the non-negative reals as our time domain: $TIME = \{\tau \in \mathbb{R} \mid \tau \geq 0\}$.

The real-time behaviour of a process P is described by the following aspects:

- the initial state, i.e., the values of the local objects at the start of the execution, and the starting time of P,
- the timed occurrence of observable actions of P, and
- if P terminates, the final state, i.e., the values of the local objects at termination, and the termination time of P (∞ if P does not terminate).

The second point, the observable behaviour of a real-time program, is modelled by a *timed occurrence function*, typically denoted by ϱ, which assigns to each point of time a set of records representing the observable events occurring at that time. To denote starting and termination times of programs, we use a special variable *now*. Then a *state*, typically denoted by $\sigma, \sigma_0, \ldots$, assigns a value from $TIME \cup \{\infty\}$ to this variable *now* and further assigns a value to each local object.

The semantics of a program P starting in a state σ_0, which is denoted by $\mathcal{M}(P)(\sigma_0)$, is a set of pairs of the form (σ, ϱ) where σ is a state and ϱ a timed occurrence function. $\sigma_0(x)$ yields the value of local object x at the start of the execution and $\sigma_0(now)$ represents the starting time. Consider a pair (σ, ϱ) in $\mathcal{M}(P)(\sigma_0)$. If P terminates then σ represents the values of the local objects at termination and $\sigma(now)$ denotes the termination time. When P does not terminate then our semantics will be such that $\sigma(now) = \infty$ and $\sigma(x)$ is an arbitrary value, for any $x \not\equiv now$. Further ϱ represents the observable behaviour of P during its execution. Thus $\varrho(\tau)$ expresses observable events of P for $\sigma_0(now) \leq \tau < \sigma(now)$. Outside this interval the occurrence of actions is not restricted by the semantics of P, so arbitrary events are allowed.

2.3. Specifications

Our specifications are based on Hoare triples (precondition, program, postcondition) [Hoa69]. To distinguish our modified triples from classical Hoare Logic, a slightly different notation will be introduced and we use the words "assumption" and "commitment" instead of, respectively, "precondition" and "postcondition". This leads to formulas of the form $\langle\!\langle A \rangle\!\rangle\ P\ \langle\!\langle C \rangle\!\rangle$, where P is a process and A and

C are assertions called, respectively, *assumption* and *commitment*.
Assertion A expresses assumptions

- about the values of local objects at the start of P,
- about the starting time of P, and
- about the timed occurrence of observable events.

Given assumption A, assertion C expresses a commitment of P,

- if P terminates, about the values of the local objects at termination,
- about the termination time (∞ if P does not terminate), and
- about the timed occurrence of observable events.

Note that, in contrast with classical Hoare triples, our commitment expresses properties of terminating as well as nonterminating computations. Together with the addition of time this means that our formalism is not restricted to partial correctness, but also deals with progress properties.

The assertions A and C in a correctness formula $\langle\langle A\rangle\rangle\ P\ \langle\langle C\rangle\rangle$ are expressed in a first-order logic with the following primitives.

- Names denoting local objects, such as $x, y, \ldots$, ranging over *VAL*.
- Logical variables that are not affected by program execution. We have logical value variables ranging over *VAL*, such as v, v_0, v_1, $\ldots$, and logical time variables ranging over $TIME \cup \{\infty\}$, such as t, t_0, $t_1, \ldots$.
- A special variable *now*, ranging over $TIME \cup \{\infty\}$, which refers to our global notion of time. An occurrence of *now* in assumption A represents the starting time of the statement P whereas in commitment C it denotes the termination time (using $now = \infty$ for nonterminating computations).
- For observable action O and expression *texp*, which yields a value in *TIME*, we use a boolean primitive O **at** *texp* to denote that O occurs at time *texp*.

Let $loc(p)$ be the set of names of local objects occurring in assertion p. Similarly, $obs(p)$ denotes the set of observables occurring in p. We use time intervals such as $[t_0, t_1) = \{t \in TIME \mid t_0 \leq t < t_1\}$, $(t_0, t_1) = \{t \in TIME \mid t_0 < t < t_1\}$, etc. Henceforth we use $\equiv$ to denote syntactic equality.
Given P **at** t and a set (usually an interval) $I \subseteq TIME$, we use

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

The notation $p[exp/var]$ is used to represent the substitution of expression *exp* for each free occurrence of variable *var* in assertion p. We assume the usual properties of ∞ such as, for all $t \in TIME$, $t < \infty$, $t + \infty = \infty + t = \infty - t = \infty$.

2.3.1. Interpretation

To interpret logical variables we use a logical variable environment γ, that is, a mapping which assigns a value from *VAL* to each logical value variable and a value from $TIME \cup \{\infty\}$ to each logical time variable. First we define the value of expression *exp* in an environment γ, a state σ, and a mapping ϱ, denoted by $\mathcal{V}(exp)(\sigma, \varrho, \gamma)$. A few illustrative cases:

- $\mathcal{V}(t)(\sigma, \varrho, \gamma) = \gamma(t)$
- $\mathcal{V}(now)(\sigma, \varrho, \gamma) = \sigma(now)$

- $\mathscr{V}(x)(\sigma, \varrho, \gamma) = \sigma(x)$
- $\mathscr{V}(exp_1 + exp_2)(\sigma, \varrho, \gamma) = \mathscr{V}(exp_1)(\sigma, \varrho, \gamma) + \mathscr{V}(exp_2)(\sigma, \varrho, \gamma)$

Next we define inductively when an assertion p holds in a triple $(\sigma, \varrho, \gamma)$, which is denoted by $(\sigma, \varrho, \gamma) \models p$. For instance,

- $(\sigma, \varrho, \gamma) \models p_1 \vee p_2$ iff $(\sigma, \varrho, \gamma) \models p_1$ or $(\sigma, \varrho, \gamma) \models p_2$.

To define the formal interpretation of a correctness formula $\langle\langle A \rangle\rangle$ P $\langle\langle C \rangle\rangle$, first observe that assumption A might refer to points in time after the starting time. Thus A might contain assumptions about the occurrence of actions *during* the execution of P. For instance, $\langle\langle now = 4 \wedge O \textbf{ at } 5 \rangle\rangle$ P $\langle\langle now = 7 \wedge O \textbf{ at } 5 \rangle\rangle$ should be valid. Therefore in the formal definition of a triple $\langle\langle A \rangle\rangle$ P $\langle\langle C \rangle\rangle$ we use the same occurrence function to interpret A and C. Further this occurrence function should correspond to the execution of P between the start and the termination of P, as represented by the semantics of P.

Definition 2.1. (Validity) For a program P and assertions A and C, a correctness formula $\langle\langle A \rangle\rangle$ P $\langle\langle C \rangle\rangle$ is *valid*, denoted by $\models \langle\langle A \rangle\rangle$ P $\langle\langle C \rangle\rangle$, iff for any γ, any $\sigma_0 \in STATE$, and any σ, ϱ with $(\sigma, \varrho) \in \mathscr{M}(S)(\sigma_0)$, we have that $(\sigma_0, \varrho, \gamma) \models A$ implies $(\sigma, \varrho, \gamma) \models C$.

2.3.2. Examples of Specifications

We give a number of small examples to illustrate our specification formalism. The first triple expresses that program F which starts at time 6 in a state where local object x has the value 5 and assuming that there is some observable action O at 3, terminates between 15 and 23 in a state where x has the value $f(5)$. Further, given the assumption, we have the commitment that O occurs at 3.

$$\begin{array}{l} \langle\langle x = 5 \wedge now = 6 \wedge O \textbf{ at } 3 \rangle\rangle \\ \quad F \\ \langle\langle x = f(5) \wedge 15 < now < 23 \wedge O \textbf{ at } 3 \rangle\rangle. \end{array}$$

We can generalize specifications by using logical variables to represent the initial values of program variables and the starting time. For instance, to specify that a program *FUN* should compute $f(x)$ within certain time bounds, for a particular function f, leaving x invariant, we can use logical variables v and t:

$$\langle\langle x = v \wedge now = t < \infty \rangle\rangle \; FUN \; \langle\langle y = f(v) \wedge x = v \wedge t + 5 < now < t + 13 \rangle\rangle.$$

Note that logical variables, such as v and t, are implicitly universally quantified. Also the real-time communication interface of nonterminating programs can be specified. For instance, a process which sends output periodically.

$$\langle\langle x = 0 \wedge now = 0 \rangle\rangle \;\; L \;\; \langle\langle now = \infty \wedge \forall i \in \mathbf{N} : (output, f(i)) \textbf{ at } T(i) \rangle\rangle.$$

Finally consider a program *REACT* with terminating as well as nonterminating computations; it terminates iff it receives input 0.

$$\begin{array}{l} \langle\langle now = 0 \rangle\rangle \\ \quad REACT \\ \langle\langle (\forall t < now : (input, v) \textbf{ at } t \rightarrow (output, f(v)) \textbf{ in } [t + T_l, t + T_u]) \wedge \\ \;\; (now < \infty \leftrightarrow \exists t_0 < now : (input, 0) \textbf{ at } t_0) \rangle\rangle. \end{array}$$

Finally note that a classical Hoare triple $\{p\}$ P $\{q\}$, denoting partial correctness, can be expressed in our framework as

$$\langle\!\langle p \wedge now < \infty \rangle\!\rangle \; P \; \langle\!\langle now < \infty \rightarrow q \rangle\!\rangle.$$

Total correctness of P with respect to p and q can be denoted by

$$\langle\!\langle p \wedge now < \infty \rangle\!\rangle \; P \; \langle\!\langle now < \infty \wedge q \rangle\!\rangle.$$

2.4. Proof Rules

The proof system contains a consequence rule which is identical to the classical rule for Hoare triples. It allows us to strengthen assumptions and to weaken commitments.

Rule 2.1. (Consequence)

$$\frac{\langle\!\langle A_0 \rangle\!\rangle \; P \; \langle\!\langle C_0 \rangle\!\rangle, A \rightarrow A_0, C_0 \rightarrow C}{\langle\!\langle A \rangle\!\rangle \; P \; \langle\!\langle C \rangle\!\rangle}$$

The proof rule for parallel composition has the following general form, using a combinator *Comb* of assertions which will be defined below.

Rule 2.2. (Parallel Composition)

$$\frac{\langle\!\langle A_1 \rangle\!\rangle \; P_1 \; \langle\!\langle C_1 \rangle\!\rangle, \quad \langle\!\langle A_2 \rangle\!\rangle \; P_2 \; \langle\!\langle C_2 \rangle\!\rangle, \quad Comb(C_1, C_2) \rightarrow C}{\langle\!\langle A_1 \wedge A_2 \rangle\!\rangle \; P_1 \| P_2 \; \langle\!\langle C \rangle\!\rangle}$$

provided

- $loc(C_1) \cap loc(P_2) = \emptyset$ and $loc(C_2) \cap loc(P_1) = \emptyset$, that is, the commitment of one process should not refer to local objects of the other.
- $obs(A_1, C_1) \cap obs(P_2) \subseteq obs(P_1)$ and $obs(A_2, C_2) \cap obs(P_1) \subseteq obs(P_2)$, i.e., if assertions in the specification of one process refer to the interface of another process then this concerns a joint interface.

We consider three possibilities for *Comb*:

1. If *now* does not occur in C_1 and C_2 then define

   $Comb(C_1, C_2) \equiv C_1 \wedge C_2.$

   Note that without the additional restricition on *now* the rule is not sound: for instance, $\langle\!\langle now = 0 \rangle\!\rangle \; P_1 \; \langle\!\langle now = 2 \rangle\!\rangle$ and $\langle\!\langle now = 0 \rangle\!\rangle \; P_2 \; \langle\!\langle now = 3 \rangle\!\rangle$ would lead to $\langle\!\langle now = 0 \rangle\!\rangle \; P_1 \| P_2 \; \langle\!\langle now = 2 \wedge now = 3 \rangle\!\rangle$ and hence, by the consequence rule, $\langle\!\langle now = 0 \rangle\!\rangle \; P_1 \| P_2 \; \langle\!\langle false \rangle\!\rangle$.
   Henceforth we refer to this version as the Simple Parallel Composition Rule.
2. The problem with *now* in the commitments is that, in general, the termination times of P_1 and P_2 will be different. To obtain a general rule, we substitute logical variables t_1 and t_2 for *now* in, respectively, C_1 and C_2. Then the termination time of $P_1 \| P_2$, expressed by *now* in its commitment, is the maximum of t_1 and t_2.

   $Comb(C_1, C_2) \equiv C_1[t_1/now] \wedge C_2[t_2/now] \wedge now = max(t_1, t_2).$
3. The definition of *Comb* above leads to a sound rule, but for completeness we need predicates to express that process P_i does not perform any action after its termination, for $i = 1, 2$. Define for a finite set *oset* of observables,

   $NoAct(oset)$ **at** $texp \equiv \bigwedge_{O \in oset} \neg O$ **at** $texp$.

Then define

$$\begin{aligned} Comb(C_1, C_2) \equiv\ & C_1[t_1/now] \wedge NoAct(obs(P_1)) \textbf{ during } [t_1, now) \wedge \\ & C_2[t_2/now] \wedge NoAct(obs(P_2)) \textbf{ during } [t_2, now) \wedge \\ & now = max(t_1, t_2). \end{aligned}$$

Note that the parallel composition rule is compositional, since a specification of the compound construct $P_1 \| P_2$ can be derived using only the specifications of the components P_1 and P_2 and their static interface as given by *loc* and *obs*.

3. Example Chemical Batch Processing

We consider a chemical batch processing example which is inspired by a description in [ALF93]. It consists of a batch processing plant which has a reaction vessel filled with chemicals. Heating two chemicals produces a third chemical which is hazardous and might lead to an explosion. To formalize the situation, we use the primitive

- expl **at** *texp* to denote that an explosion occurs in the vessel at time *texp*.

Define $obs(\mathsf{expl}\ \textbf{at}\ texp) = \{\mathsf{expl}\}$.
The top-level specification of the chemical batch processing system *CBP* requires that there should be no explosion.

$$\langle\langle now = 0 \rangle\rangle\ CBP\ \langle\langle \forall t < \infty : \neg \mathsf{expl}\ \textbf{at}\ t \rangle\rangle.$$

To implement a control system which establishes this property, we first model a physical property of the chemicals in the vessel *V*. We use

- empty **at** *texp* to denote that the vessel is empty at time *texp*;
- temp(*texp*) to denote the temperature inside the vessel at time *texp*.

Define $obs(\mathsf{empty}\ \textbf{at}\ texp) = \{\mathsf{empty}\}$ and $obs(\mathsf{temp}(texp)) = \{\mathsf{temp}\}$.
Suppose some chemical analysis yields that there will be no explosion if the temperature is below a certain value, say ExpTemp, or if the vessel is empty. Let

$$CV \equiv \forall t < \infty : \mathsf{temp}(t) \leq \mathsf{ExpTemp} \vee \mathsf{empty}\ \textbf{at}\ t \rightarrow \neg \mathsf{expl}\ \textbf{at}\ t.$$

Then, the vessel *V* is specified by

$$\langle\langle now = 0 \rangle\rangle\ V\ \langle\langle CV \rangle\rangle$$

with $obs(V) = \{\mathsf{expl}, \mathsf{temp}, \mathsf{empty}\}$, where expl and empty are boolean variables and temp a variable ranging over $\mathbb{R}$.

Given this property, there are several possible strategies for a control system when it detects that the temperature is too high. For instance it might cool the chemicals while they are in the vessel. Here we follow [ALF93] and decide to empty the contents into a cooled vat. Let

$$CHL \equiv \forall t < \infty : \mathsf{temp}(t) > \mathsf{ExpTemp} \rightarrow \mathsf{empty}\ \textbf{at}\ t.$$

Then a high-level specification of the control component *HLContr* is given by

$$\langle\langle now = 0 \rangle\rangle\ HLContr\ \langle\langle CHL \rangle\rangle$$

with $obs(HLContr) \supseteq \{\mathsf{temp}, \mathsf{empty}\}$, that is, the high-level control component can observe the temperature and whether the vessel is empty.

Observe that $obs(CV) \cap obs(HLContr) \subseteq obs(CV) = \{\mathsf{expl}, \mathsf{temp}, \mathsf{empty}\} = obs(V)$ and $obs(CHL) \cap obs(V) = \{\mathsf{temp}, \mathsf{empty}\} \subseteq obs(HLContr)$. Hence, since there are no local objects, the specifications of V and $HLContr$ satisfy the requirements of the Simple Parallel Composition Rule and we obtain

$$\langle\langle now = 0 \rangle\rangle \; V \| HLContr \; \langle\langle CV \wedge CHL \rangle\rangle.$$

Clearly the commitment $CV \wedge CHL$ implies $\forall t < \infty : \neg \mathsf{expl}$ **at** t. Thus, by the consequence rule, we obtain that $V \| HLContr$ is a correct implementation of CBP. Considering the specification of V as given, it remains to implement $HLContr$ according to its specification. Therefore we first introduce a communication mechanism in the next section, and then continue with the example in Section 5.

4. Asynchronous Communication

In this paper we consider parallel processes that communicate via message passing along unidirectional channels. Channels are point-to-point, i.e. each connecting two processes, and asynchronous, that is, a sender does not wait for a receiver, but sends immediately. There is no buffering of messages; the message is lost if there is no receiver. A receiving process waits until a message is available.

Let *CHAN* be a nonempty set of channel names. To describe communication by message passing along asynchronous channels, the assertion language contains the following primitives, where $c \in CHAN$, and *exp* and *texp* are expressions yielding a value in *VAL* and *TIME*, respectively.

- $(c!!, exp)$ **at** *texp* to denote that a process starts sending value *exp* along channel c at time *texp*.
- $c?$ **at** *texp* to express that a process is waiting to receive a message along channel c at time *texp*.
- (c, exp) **at** *texp* to denote that a process starts to receive value *exp* along channel c at time *texp*.

Define for these primitives $obs((c!!, exp)$ **at** $texp) = \{c!!\}$, $obs(c?$ **at** $exp) = \{c?\}$, and $obs((c, exp)$ **at** $texp) = \{c\}$.

Next we define a few useful abbreviations. To expresses that a process starts waiting to receive input along c at time t until it receives input with value v, allowing the possibility that it has to wait forever, we introduce

- $await\,(c?, v)$ **at** $t \equiv c?$ **during** $[t, \infty) \vee$
  $(\exists t_1 \in [t, \infty) : c?$ **during** $[t, t_1) \wedge (c, v)$ **at** $t_1)$

We often abstract from the value that is transmitted, using

- c **at** $t \equiv \exists v : (c, v)$ **at** t
- $c!!$ **at** $t \equiv \exists v : (c!!, v)$ **at** t
- $await\; c?$ **at** $t \equiv \exists v : await\,(c?, v)$ **at** t

Similar abbreviations can be defined with general expressions instead of v and t. Further we will sometimes use $(P_1 \wedge P_2)$ **at** t instead of P_1 **at** $t \wedge P_2$ **at** t, etc.

4.1. Communication Properties

We axiomatize three properties of asynchronous channels. First, at any point of time at most one message is transmitted on any channel c, that is,

- $\forall t < \infty \, \forall v_1, v_2 : (c!!, v_1) \textbf{ at } t \wedge (c!!, v_2) \textbf{ at } t \rightarrow v_1 = v_2$

As mentioned before, we adopt the maximal parallelism model in this paper. This implies *minimal waiting*, that is, a process only waits when it has to receive input and no message is available. Further, for simplicity, we assume that when a process starts sending a message it is immediately available for the receiver. Then a process can only receive a message along a channel c if this message is transmitted simultaneously, i.e.,

- $\forall t < \infty \, \forall v : (c, v) \textbf{ at } t \rightarrow (c!!, v) \textbf{ at } t$

Minimal waiting is expressed by stating that no process should be waiting to receive along channel c if a message is transmitted (and hence available) on c:

- $\forall t < \infty : \neg(c!! \textbf{ at } t \wedge c? \textbf{ at } t)$

Note It is not difficult to adapt the framework to more realistic assumptions. Suppose, e.g., that it takes Δ time units before a message transmitted by a sender is available for a receiver. Then the second property becomes

$$\forall t < \infty \, \forall v : (c, v) \textbf{ at } t \rightarrow t \geq \Delta \wedge (c!!, v) \textbf{ at } (t - \Delta)$$

and we should change the third property into

$$\forall t < \infty : \neg(c!! \textbf{ at } t \wedge c? \textbf{ at } (t + \Delta)).$$

Another option is to assume that an output is available during a certain period, and several inputs can read the same output. (Similar to a shared variable.) Then the second and third property should become, respectively,

$$\forall t < \infty \, \forall v : (c, v) \textbf{ at } t \rightarrow$$
$$\exists t_0 \in [t - \Delta_1, t - \Delta_2] : (c!!, v) \textbf{ at } t_0 \wedge \, (\neg c!!) \textbf{ during } (t_0, t - \Delta_2]$$

$$\forall t < \infty : \neg(c!! \textbf{ at } t \wedge c? \textbf{ during } [t + \Delta_1, t + \Delta_2])$$

End Note Based on these properties we give a few useful lemmas.The first lemma expresses that if the sender does not send before Δ_1 and with a distance of at least Δ_2, and if the receiver is ready to receive before Δ_1 and with a distance of at most Δ_2, then each output is received, i.e., no message gets lost. Define

$maxsend(c, \Delta_1, \Delta_2) \textbf{ at } t \equiv c!! \textbf{ at } t \rightarrow t \geq \Delta_1 \wedge (\neg c!!) \textbf{ during } (t - \Delta_2, t)$.

$minwait(c, \Delta_1, \Delta_2) \textbf{ at } t \equiv t \geq \Delta_1 \rightarrow await\ c? \textbf{ in } (t - \Delta_2, t]$.

Lemma 4.1. If $maxsend(c, \Delta_1, \Delta_2) \textbf{ during } [0, \infty)$, and $minwait(c, \Delta_1, \Delta_2) \textbf{ during } [0, \infty)$, then $\forall t < \infty : c!! \textbf{ at } t \leftrightarrow c \textbf{ at } t$.

Proof. By the communication property mentioned above, $c \textbf{ at } t \rightarrow c!! \textbf{ at } t$. Hence it remains to prove $\forall t < \infty : c!! \textbf{ at } t \rightarrow c \textbf{ at } t$. Suppose $c!! \textbf{ at } t$. With assumption $maxsend(c, \Delta_1, \Delta_2) \textbf{ during } [0, \infty)$ this leads to $t \geq \Delta_1$ and $(\neg c!!) \textbf{ during } (t - \Delta_2, t)$. Hence by the communication property mentioned before, we obtain that $(\neg c) \textbf{ during } (t - \Delta_2, t)$. Since $t \geq \Delta_1$, the assumption $minwait(c, \Delta_1, \Delta_2) \textbf{ during } [0, \infty)$ leads to $await\ c? \textbf{ in } (t - \Delta_2, t]$. With $(\neg c) \textbf{ during } (t - \Delta_2, t)$ this implies $await\ c? \textbf{ at } t$. By $c!! \textbf{ at } t$, and the minimal waiting property this leads to $c \textbf{ at } t$. □

The next lemma expresses that if the sender sends regularly, e.g. at least once every Δ_s time units, and the receiver is ready to receive at least once every Δ_r time units, then there is a communication at least once every $\Delta_s + \Delta_r$ time units.

Lemma 4.2. If $\forall t < \infty : c!! \textbf{ in } [t, t + \Delta_s)$ and $\forall t < \infty : \mathit{await}\ c? \textbf{ in } [t, t + \Delta_r)$ then $\forall t < \infty : c \textbf{ in } [t, t + \Delta_s + \Delta_r)$.

Proof. Consider $t < \infty$. Then $\mathit{await}\ c? \textbf{ in } [t, t + \Delta_r)$. If $c \textbf{ in } [t, t + \Delta_r)$ then clearly $c \textbf{ in } [t, t + \Delta_s + \Delta_r)$. If $(\neg c) \textbf{ during } [t, t + \Delta_r)$, then $\mathit{await}\ c? \textbf{ at } t + \Delta_r$, and the assumption $c!! \textbf{ in } [t + \Delta_r, t + \Delta_r + \Delta_s)$ leads to $c \textbf{ in } [t + \Delta_r, t + \Delta_r + \Delta_s)$. □

5. Introducing Sensor and Actuator in the Example

In this section we refine process *HLContr* of Section 3 specified by

$$\langle\langle now = 0 \rangle\rangle\ HLContr\ \langle\langle CHL \rangle\rangle$$

where $CHL \equiv \forall t < \infty : \mathsf{temp}(t) > \mathsf{ExpTemp} \rightarrow \mathsf{empty} \textbf{ at } t$, and with $obs(HLContr) \supseteq \{\mathsf{temp}, \mathsf{empty}\}$.

First, in Section 5.1 we introduce a sensor to measure the temperature. Next, in Section 5.2, we use an actuator to empty the vessel.

5.1. Sensor

Suppose we have a thermometer *Therm* which measures the temperature and sends the measured values along channel thermchan. We assume that the value of the thermometer does not deviate more than ThermDev from the real temperature. Here we only need an upper bound and postulate

$CSEN_1 \equiv \forall t < \infty\, \forall v : (\mathsf{thermchan}!!, v) \textbf{ at } t \rightarrow \mathsf{temp}(t) - v < \mathsf{ThermDev}$.

The thermometer will send values at least once every DelTherm time units:

$CSEN_2 \equiv \forall t < \infty : \mathsf{thermchan}!! \textbf{ in } [t, t + \mathsf{DelTherm})$.

Further there are two assumptions about the maximal change of the temperature, using a parameter MaxRise > 0, and the initial temperature, using a safety temperature SafeTemp.

$CSEN_3 \equiv \forall t_0 < \infty\, t_1 < \infty : \mathsf{temp}(t_1) - \mathsf{temp}(t_0) < \mathsf{MaxRise} \times (t_1 - t_0)$.

$CSEN_4 \equiv \mathsf{temp}(0) \leq \mathsf{SafeTemp} + \mathsf{ThermDev}$.

Let $CSEN \equiv CSEN_1 \wedge CSEN_2 \wedge CSEN_3 \wedge CSEN_4$. Then assume

$$\langle\langle now = 0 \rangle\rangle\ Therm\ \langle\langle CSEN \rangle\rangle$$

with $obs(Therm) = \{\mathsf{temp}, \mathsf{thermchan}!!\}$.

To obtain the specification of *HLContr*, we design in parallel with the thermometer a flow control component *FlowContr* with the following commitments. The control component is ready to receive input from the sensor along thermchan at least once every DelReadTherm time units.

$CFC_1 \equiv \forall t < \infty : \mathit{await}\ \mathsf{thermchan}? \textbf{ in } [t, t + \mathsf{DelReadTherm})$.

Next we specify that when an unsafe temperature is detected, i.e. above safety temperature SafeTemp, the vessel is emptied in at most DelEmpty time units. Let *EmptyVessel* **at** $t \equiv$ empty **at** $[t, \infty)$.

$$CFC_2 \equiv \forall t < \infty \, \forall v : (\text{thermchan}, v) \textbf{ at } t \wedge v > \text{SafeTemp} \rightarrow \\ \qquad \textit{EmptyVessel} \textbf{ in } [t, t + \text{DelEmpty}].$$

Define $CFC \equiv CFC_1 \wedge CFC_2$, and specify

$$\langle\langle now = 0 \rangle\rangle \; FlowContr \; \langle\langle CFC \rangle\rangle$$

Note that the syntactic requirements of the Simple Parallel Composition Rule are fulfilled: $obs(CSEN) \cap obs(FlowContr) \subseteq \{\text{temp}, \text{thermchan!!}\} = obs(Therm)$ and $obs(CFC) \cap obs(Therm) = \emptyset \subseteq obs(FlowContr)$. Hence we obtain

$$\langle\langle now = 0 \rangle\rangle \; Therm \| FlowContr \; \langle\langle CSEN \wedge CFC \rangle\rangle.$$

Define TempDiff = ExpTemp − (SafeTemp + ThermDev).

Lemma 5.1. If TempDiff $\geq$ MaxRise $\times$ (DelTherm + DelReadTherm + DelEmpty) then $CSEN \wedge CFC \rightarrow CHL$.

Proof. Assume temp$(t) >$ ExpTemp. We have to show empty **at** t. Observe that this follows from *EmptyVessel* **in** $[t_1, t_1 + \text{DelEmpty}]$ for some t_1 if

$$t_1 + \text{DelEmpty} \leq t \qquad \text{(req1)}$$

By CFC_2, this can be derived if, for some v,

$$(\text{thermchan}, v) \textbf{ at } t_1 \qquad \text{(req2)}$$
$$v > \text{SafeTemp} \qquad \text{(req3)}$$

Note that by $CSEN_2$ and CFC_1 we obtain, using lemma 4.2,

$$\forall t_0 < \infty : \text{thermchan} \textbf{ in } [t_0, t_0 + \text{DelTherm} + \text{DelReadTherm}).$$

Hence thermchan **in** $[t - \text{DelTherm} - \text{DelReadTherm} - \text{DelEmpty}, t - \text{DelEmpty})$ provided

$$t - \text{DelTherm} - \text{DelReadTherm} - \text{DelEmpty} \geq 0 \qquad \text{(req4)}$$

Thus there exists a $t_1 \in [t - \text{DelTherm} - \text{DelReadTherm} - \text{DelEmpty}, t - \text{DelEmpty})$ and a v satisfying (req1) and (req2). Note that, since $t_1 \geq t - \text{DelTherm} - \text{DelReadTherm} - \text{DelEmpty}$, we have $t - t_1 \leq \text{DelTherm} + \text{DelReadTherm} + \text{DelEmpty}$. To prove requirement (req3), observe that $(\text{thermchan}, v)$ **at** t_1 implies $(\text{thermchan!!}, v)$ **at** t_1, and thus by $CSEN_1$, $v > temp(t_1) - \text{ThermDev}$. Hence it remains to prove $temp(t_1) > \text{SafeTemp} + \text{ThermDev}$.

By $CSEN_3$, $\text{temp}(t) - \text{temp}(t_1) < \text{MaxRise} \times (t - t_1)$, thus (using MaxRise > 0), $temp(t_1) > temp(t) - \text{MaxRise} \times (t - t_1) > \text{ExpTemp} - \text{MaxRise} \times (\text{DelTherm} + \text{DelReadTherm} + \text{DelEmpty}) \geq \text{ExpTemp} - \text{TempDiff} = \text{ExpTemp} - (\text{ExpTemp} - (\text{SafeTemp} + \text{ThermDev})) = \text{SafeTemp} + \text{ThermDev}$.

It remains to prove (req4). By $CSEN_3$, $\text{temp}(t) - \text{temp}(0) < \text{MaxRise} \times (t - 0)$, thus $t > (\text{temp}(t) - \text{temp}(0))/\text{MaxRise}$. Using temp$(t) >$ ExpTemp and $CSEN_4$ this leads to $t > (\text{ExpTemp} - (\text{SafeTemp} + \text{ThermDev}))/\text{MaxRise} = \text{TempDiff}/\text{MaxRise}$. Hence $t - \text{DelTherm} - \text{DelReadTherm} - \text{DelEmpty} \geq \text{TempDiff}/\text{MaxRise} - \text{DelTherm} - \text{DelReadTherm} - \text{DelEmpty} \geq 0$. □

Hence, by lemma 5.1, the consequence rule leads to

$$\langle\langle now = 0 \rangle\rangle \; Therm \| FlowContr \; \langle\langle CHL \rangle\rangle.$$

Assuming $obs(FlowContr) \supseteq \{\mathsf{empty}\}$ we obtain that $obs(Therm \| FlowContr) \supseteq \{\mathsf{temp}, \mathsf{empty}\}$, and thus $Therm \| FlowContr$ is a correct refinement of $HLContr$. We continue with the implementation of $FlowContr$.

5.2. Actuator

In order to implement $FlowContr$ we use an actuator to empty the vessel. Suppose the actuator can be activated by sending a message along channel actchan. First we assume that the actuator is ready to receive input periodically, using parameters Init and Period, (recall that *minwait* has been defined in Section 4)

$CA_1 \equiv minwait(\mathsf{actchan}, \mathsf{Init}, \mathsf{Period})$ **during** $[0, \infty)$.

Further the actuator will respond to a signal along actchan by emptying the vessel in at most DelAct time units.

$CA_2 \equiv \forall t < \infty : \mathsf{actchan}$ **at** $t \rightarrow EmptyVessel$ **in** $[t, t + \mathsf{DelAct}]$.

Let $CA \equiv CA_1 \wedge CA_2$ and assume

$\langle\langle now = 0 \rangle\rangle \; Actuator \; \langle\langle CA \rangle\rangle$

with $obs(Actuator) = \{\mathsf{actchan?}, \mathsf{actchan}, \mathsf{empty}\}$.

In parallel with this actuator we design a control component $Contr$ which sends signals to the actuator along actchan. To guarantee that no message is lost we specify, in view of CA_1, the maximal frequency with which it will send messages along actchan.

$CC_1 \equiv maxsend(\mathsf{actchan}, \mathsf{Init}, \mathsf{Period})$ **during** $[0, \infty)$.

The main task of the control component is to send a signal along actchan, in at most DelContr time units, if it receives a value via thermchan which is greater than SafeTemp.

$CC_2 \equiv \forall t < \infty \, \forall v : (\mathsf{thermchan}, v)$ **at** $t \wedge v > \mathsf{SafeTemp} \rightarrow$
$\mathsf{actchan!!}$ **in** $[t, t + \mathsf{DelContr}]$.

Define $CC \equiv CC_1 \wedge CC_2 \wedge CFC_1$, and specify

$\langle\langle now = 0 \rangle\rangle \; Contr \; \langle\langle CC \rangle\rangle$

Observe that $obs(CA) \cap obs(Contr) \subseteq obs(CA) = \{\mathsf{actchan?}, \mathsf{actchan}, \mathsf{empty}\} = obs(Actuator)$ and $obs(CC) \cap obs(Actuator) = \emptyset \subseteq obs(Contr)$. Then the Simple Parallel Composition Rule leads to

$\langle\langle now = 0 \rangle\rangle \; Actuator \| Contr \; \langle\langle CA \wedge CC \rangle\rangle$.

Lemma 5.2. If $\mathsf{DelEmpty} = \mathsf{DelAct} + \mathsf{DelContr}$ then $CA \wedge CC \rightarrow CFC$.

Proof. To prove CFC, it remains to prove CFC_2.
Suppose $(\mathsf{thermchan}, v)$ **at** t and $v > \mathsf{SafeTemp}$.
By CC_2 we obtain $\mathsf{actchan!!}$ **in** $[t, t + \mathsf{DelContr}]$.
Note that by CA_1 and CC_1 we obtain, using lemma 4.1,
$\forall t_0 < \infty : \mathsf{actchan!!}$ **at** $t_0 \leftrightarrow \mathsf{actchan}$ **at** t_0.
Hence actchan **in** $[t, t + \mathsf{DelContr}]$.

Thus CA_2 leads to *EmptyVessel* **in** $[t, t + \mathsf{DelAct} + \mathsf{DelContr}]$, and hence *EmptyVessel* **in** $[t, t + \mathsf{DelEmpty}]$, using $\mathsf{DelEmpty} = \mathsf{DelAct} + \mathsf{DelContr}$. □

Note that $obs(Actuator \| Contr) \supseteq obs(Actuator) \supseteq \{\mathsf{empty}\}$. Hence, by the consequence rule and the lemma above, $Actuator \| Contr$ refines $FlowContr$.

Concluding, the design steps in this section have been proved correct provided $\mathsf{MaxRise} > 0$ and $\mathsf{ExpTemp}-(\mathsf{SafeTemp}+\mathsf{ThermDev}) \geq \mathsf{MaxRise} \times (\mathsf{DelTherm}+\mathsf{DelReadTherm}+\mathsf{DelAct}+\mathsf{DelContr})$. It remains to implement *Contr*, which will be done in section 7 after the introduction of a real-time programming language in the next section.

6. Programming Language

To program distributed real-time systems we present in Section 6.1 syntax and informal semantics of a real-time programming language. The basic timing assumptions are given in Section 6.2. To show that programs satisfy an assumption/commitment specification we formulate in Section 6.3 a compositional proof system. We do not give a formal semantics of the programming language in this paper, but the essential ideas are reflected in the proof system of Section 6.3. Details about the formulation of a denotational semantics for a real-time programming language are given in [Hoo91], where also proofs of soundness and completeness can be found.

6.1. Syntax Programming Language

We consider a real-time concurrent programming language with communication along asynchronous channels. Real-time is incorporated by delay-statements which suspend the execution for a certain period of time. Such a delay-statement is also allowed in a simple select statement (similar to a delay-statement in the select-construct of Ada [Ada83]).

The statements of the programming language and their informal meaning, are listed below, using program variable x, expression e yielding a value in VAL, boolean expression b, and $c \in CHAN$.

Atomic statements

- **skip** terminates immediately.
- Assignment $x := e$ assigns the value of expression e to the variable x.
- **delay** e suspends execution for (the value of) e time units. If e yields a negative value then **delay** e is equivalent to **skip**.
- Output statement $c!!e$ is used to send the value of expression e along channel c. It does not wait for a receiver but sends immediately.
- Input statement $c?x$ is used to receive a value along channel c and assign this value to the variable x. Such an input statement has to wait until a message is available.

Compound statements

- $S_1 ; S_2$ indicates sequential composition.
- **if** b **then** S_1 **else** S_2 **fi** denotes the usual conditional choice construct.

- **sel** $c?x$ **then** S_1 **or delay** e **then** S_2 **les** is a select statement. First wait to receive a message on channel c and, if a message is available within e time units, execute S_1. If no message is available during e time units, S_2 is executed.

  Example 6.1. By restricting the waiting period for an input statement, this construct represents a *time-out.* Consider for instance

  sel $in?x$ **then** $out!!f(x)$ **or delay** 8 **then** $alarm!!y$ **les**.

  This statement waits to receive a message along channel *in* during at most 8 time units, and if no message is available within 8 time units a message is sent along channel *alarm.*

- **while** b **do** S **od**, the traditional while statement.
- $S_1 \| S_2$ indicates parallel execution of the statements S_1 and S_2. We require that S_1 and S_2 do not have shared variables. The components S_1 and S_2 of a parallel composition are often called *processes.*

We also use **if** b **then** S **fi** as an abbreviation of **if** b **then** S **else skip fi**.

Let $loc(S)$ be the set of program variables occurring in S. We define $obs(S)$, the set of observables of S, by induction on the structure of S. The main clauses are $obs(c!!e) = \{c!!\}$ and $obs(c?x) = \{c?, c\}$. The rest is straightforward: $obs(\textbf{skip}) = obs(x := e) = obs(\textbf{delay}\ e) = \emptyset$, $obs(S_1;S_2) = obs(S_1\|S_2) = obs(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}) = obs(\textbf{sel}\ c?x\ \textbf{then}\ S_1\ \textbf{or delay}\ e\ \textbf{then}\ S_2\ \textbf{les}) = \{c?, c\} \cup obs(S_1) \cup obs(S_2)$, and $obs(\textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}) = obs(S)$.

Observe that for $S_1\|S_2$ we have $loc(S_1) \cap loc(S_2) = \emptyset$ and, since channels are unidirectional and connect at most two processes, $obs(S_1) \cap obs(S_2) = \emptyset$.

6.2. Basic Timing Assumptions

To determine the timing behaviour of programs we have to make assumptions about the execution time needed for the atomic constructs and how the execution time of compound constructs can be obtained from the timing of the components. Note that the maximal parallelism model adopted here implies *maximal progress,* i.e. an enabled action will be executed as soon as possible. A process never waits with the execution of a local, non-communication, command or with the execution of an asynchronous output. An input command can cause a process to wait, but only when no message is available. Thus, to describe the real-time behaviour of programs, we have to make assumptions about:

- The execution time of atomic statements. Here we use (nonnegative) parameters representing the duration of atomic statements. We assume that
  - there exists a parameter T_a such that each assignment of the form $x := e$ takes T_a time units;
  - **delay** e takes exactly e time units if e is positive and 0 time units otherwise;
  - there exist a parameter $T_{comm} > 0$ such that each communication takes T_{comm} time units.
- The extra time required to execute compound programming constructs. Here we assume that there exists a parameter T_w such that the evaluation of the boolean b in a while construct **while** b **do** S **od** takes T_w time

units. To avoid an infinite loop in finite time, we assume that T_w has a fixed positive lower bound. This guarantees finite variability (also called nonZenoness).

6.3. Proof System Programming Language

We formulate a compositional proof system for our extended Hoare logic. First we give rules and axioms that are generally applicable to any statement. Next the programming language is axiomatized by giving rules and axioms for the atomic statements and the compound programming constructs. We assume that all logical variables which are introduced in the rules are fresh.

6.3.1. General Rules and Axioms

We give three axioms to deduce invariance properties. The first axiom expresses that an assumption which satisfies certain restrictions is not affected by the execution of any program.

Axiom 6.1. (Initial Invariance) $\langle\langle p \rangle\rangle\ S\ \langle\langle p \rangle\rangle$

provided p does not refer to *now* or program variables ($loc(p) = \emptyset$).

In the next invariance axiom we express that the value of any program variable which does not occur in a program S is not affected by terminating computations of S.

Axiom 6.2. (Variable Invariance) $\langle\langle p \rangle\rangle\ S\ \langle\langle now < \infty \rightarrow p \rangle\rangle$

provided *now* does not occur in p and $loc(p) \cap loc(S) = \emptyset$.

The channel invariance axiom below expresses that a program S never performs an action which does not syntactically occur in S. Recall that, for a set *oset* of observables, *NoAct*(*oset*) has been defined in Section 2.4.

Axiom 6.3. (Observables Invariance)

$$\langle\langle now = t_0 \rangle\rangle\ S\ \langle\langle NoAct(oset)\ \textbf{during}\ [t_0, now) \rangle\rangle$$

provided *oset* is a finite set of observables with $oset \cap obs(S) = \emptyset$.

Example 6.2. We give a few examples to illustrate the invariance axioms.

- By the Initial Invariance Axiom we can deduce, e.g., for any program S,
  $\langle\langle (c,5)\ \textbf{at}\ t \wedge (d!!,v)\ \textbf{at}\ (t+7) \rangle\rangle\ S\ \langle\langle (c,5)\ \textbf{at}\ t \wedge (d!!,v)\ \textbf{at}\ (t+7) \rangle\rangle$.
- The Variable Invariance Axiom allows us to derive
  $\langle\langle x = 5 \rangle\rangle\ \textbf{while}\ y \neq 0\ \textbf{do}\ c?y\ ;\ d!!f(y)\ \textbf{od}\ \langle\langle now < \infty \rightarrow x = 5 \rangle\rangle$.
  Note that for nonterminating computations it should not be possible to prove in the commitment that program variables have a particular value.
- By the Observables Invariance Axiom we can deduce, since $obs(c?x) = \{c?, c\}$,
  $\langle\langle now = t_0 \rangle\rangle\ c?x\ \langle\langle (\neg c!!)\ \textbf{during}\ [t_0, now) \wedge (\neg d)\ \textbf{during}\ [t_0, now) \rangle\rangle$.

The next axiom expresses that a program following a nonterminating computation has no effect.

Axiom 6.4. (Nontermination) $\langle\!\langle p \wedge now = \infty \rangle\!\rangle\ S\ \langle\!\langle p \wedge now = \infty \rangle\!\rangle$

The substitution rule allows us to replace a logical variable in the assumption by any arbitrary expression if this variable does not occur in the commitment.

Rule 6.1. (Substitution)

$$\frac{\langle\!\langle p \rangle\!\rangle\ S\ \langle\!\langle q \rangle\!\rangle}{\langle\!\langle p[exp/t] \rangle\!\rangle\ S\ \langle\!\langle q \rangle\!\rangle}$$

provided t does not occur free in q.

The proof system contains a conjunction rule and a disjunction rule which are identical to the classical rules for Hoare triples.

Rule 6.2. (Conjunction)

$$\frac{\langle\!\langle p_1 \rangle\!\rangle\ S\ \langle\!\langle q_1 \rangle\!\rangle, \langle\!\langle p_2 \rangle\!\rangle\ S\ \langle\!\langle q_2 \rangle\!\rangle}{\langle\!\langle p_1 \wedge p_2 \rangle\!\rangle\ S\ \langle\!\langle q_1 \wedge q_2 \rangle\!\rangle}$$

Rule 6.3. (Disjunction)

$$\frac{\langle\!\langle p_1 \rangle\!\rangle\ S\ \langle\!\langle q_1 \rangle\!\rangle, \langle\!\langle p_2 \rangle\!\rangle\ S\ \langle\!\langle q_2 \rangle\!\rangle}{\langle\!\langle p_1 \vee p_2 \rangle\!\rangle\ S\ \langle\!\langle q_1 \vee q_2 \rangle\!\rangle}$$

6.3.2. Axiomatization of Programming Constructs

A skip statement terminates immediately and has no effect.

Axiom 6.5. (Skip) $\langle\!\langle p \rangle\!\rangle$ **skip** $\langle\!\langle p \rangle\!\rangle$

In the rule for an assignment $x := e$ we express that to obtain commitment q the assumption $q[e/x, now + T_a/now] \wedge now < \infty$ is required (i.e., this is the weakest assumption). Note that, in addition to the classical rule, we also update the time to express that the termination time equals the initial time plus T_a time units.

Axiom 6.6. (Assignment) $\langle\!\langle q[e/x, now + T_a/now] \wedge now < \infty \rangle\!\rangle\ x := e\ \langle\!\langle q \rangle\!\rangle$

Example 6.3. We show that we can derive

$\langle\!\langle x = 5 \wedge now = 6 \wedge (c,0)\ \textbf{at}\ 3 \rangle\!\rangle\ x := x + 7\ \langle\!\langle x = 12 \wedge now = 6 + T_a \wedge (c,0)\ \textbf{at}\ 3 \rangle\!\rangle$.

The assignment axiom leads to

$$\begin{aligned}&\langle\!\langle x + 7 = 12 \wedge now + T_a = 6 + T_a \wedge (c,0)\ \textbf{at}\ 3 \wedge now < \infty \rangle\!\rangle\\ &\quad x := x + 7\\ &\langle\!\langle x = 12 \wedge now = 6 + T_a \wedge (c,0)\ \textbf{at}\ 3 \rangle\!\rangle.\end{aligned}$$

Then the consequence rule yields the required triple, since $x = 5 \wedge now = 6 \wedge (c,0)\ \textbf{at}\ 3$ implies $x + 7 = 12 \wedge now + T_a = 6 + T_a \wedge (c,0)\ \textbf{at}\ 3 \wedge now < \infty$.

Axiom 6.7. (Delay) $\langle\!\langle q[now + max(0,e)/now] \wedge now < \infty \rangle\!\rangle$ **delay** $e\ \langle\!\langle q \rangle\!\rangle$

In the rule for an asynchronous output action $c!!e$ first now in assumption $p \wedge now < \infty$ is replaced by t_0. Hence t_0 represents the starting time of the statement, and we express that it starts sending the value of e at time t_0, denoted by $(c!!, e)$ **at** t_0. For completeness, we also assert that no transmission is started after t_0 until it terminates, expressed by $(\neg c!!)$ **during** (t_0, now), where now represents the termination time which equals $t_0 + T_{comm}$.

Rule 6.4. (Output)

$$\frac{(p \wedge now < \infty)[t_0/now] \wedge (c!!, e) \textbf{ at } t_0 \wedge (\neg c!!) \textbf{ during } (t_0, now) \wedge now = t_0 + T_{comm} \rightarrow q}{\langle\langle p \wedge now < \infty \rangle\rangle \; c!!e \; \langle\langle q \rangle\rangle}$$

In the rule for an input statement $c?x$ we first replace *now* in the assumption $p \wedge now < \infty$ by t_0. Thus t_0 denotes the starting time. Observe that an input statement might have to wait until a message is available, i.e. a corresponding output statement is transmitting a value. To obtain a compositional proof system no assumption should be imposed upon the environment. Hence the rule should include any arbitrary waiting period (including an infinite one) and, if a comunication takes place, any arbitrary value can be received. In the rule below the commitment is split up in q_{nt}, representing a nonterminating computation with infinite waiting, i.e. $c? \textbf{ during } [t_0, \infty)$, and commitment q which expresses properties of terminating computations. In the last case there should be a point t at which a value v is received and until that point the statement is waiting to receive (thus claiming that no message was available earlier). After t until the termination time, represented by *now*, the statement does not wait or start receiving a message. To express this case, we introduce the abbreviation

$comm(c, v)(t_0, t) \equiv c? \textbf{ during } [t_0, t) \wedge (c, v) \textbf{ at } t \wedge (\neg c? \wedge \neg c) \textbf{ during } (t, now).$

The termination time equals $t + T_{comm}$ and the value v is assigned to x.

Rule 6.5. (Input)

$$\frac{\begin{array}{l}(p \wedge now < \infty)[t_0/now] \wedge c? \textbf{ during } [t_0, \infty) \wedge now = \infty \rightarrow q_{nt} \\ (p \wedge now < \infty)[t_0/now] \wedge \exists t \in [t_0, \infty) : comm(c, v)(t_0, t) \wedge now = t + T_{comm} \\ \qquad \rightarrow q[v/x]\end{array}}{\langle\langle p \wedge now < \infty \rangle\rangle \; c?x \; \langle\langle q_{nt} \vee q \rangle\rangle}$$

provided $loc(q_{nt}) = \varnothing$.

Example 6.4. By the input rule we can derive

$$\langle\langle now = 5 \rangle\rangle \;\; c?x \;\; \langle\langle (c? \textbf{ during } [5, \infty) \wedge now = \infty) \vee (\exists t \in [5, \infty) : c? \textbf{ during } [5, t) \wedge (c, x) \textbf{ at } t \wedge now = t + T_{comm}) \rangle\rangle,$$

since $t_0 = 5 \wedge c? \textbf{ during } [t_0, \infty) \wedge now = \infty \rightarrow c? \textbf{ during } [5, \infty) \wedge now = \infty$, and $t_0 = 5 \wedge \exists t \in [t_0, \infty) : comm(c, v)(t_0, t) \wedge now = t + T_{comm}$ implies
$\exists t \in [5, \infty) : c? \textbf{ during } [5, t) \wedge (c, v) \textbf{ at } t \wedge now = t + T_{comm}$, i.e.,
$(\exists t \in [5, \infty) : c? \textbf{ during } [5, t) \wedge (c, x) \textbf{ at } t \wedge now = t + T_{comm})[v/x]$.

The inference rule for sequential composition is similar to the classical rule for Hoare triples.

Rule 6.6. (Sequential Composition) $$\frac{\langle\langle p \rangle\rangle \; S_1 \; \langle\langle r \rangle\rangle, \quad \langle\langle r \rangle\rangle \; S_2 \; \langle\langle q \rangle\rangle}{\langle\langle p \rangle\rangle \; S_1 ; S_2 \; \langle\langle q \rangle\rangle}$$

Note that in contrast with classical Hoare logic, the commitment r in the rule above might describe nonterminating executions of S_1. This part of r is not affected by S_2 and can be included in q, as illustrated by the following example.

Example 6.5. Consider a program $c?y\ ;\ y := y+1$. Let $p \equiv now = 7$ and
$q \equiv (now = \infty \wedge c?\ \textbf{during}\ [7,\infty)) \vee$
$\quad (\exists t \in [7,\infty) : now = t + T_{comm} + T_a \wedge (c, y-1)\ \textbf{at}\ t)$.
To prove $\langle\!\langle p \rangle\!\rangle\ c?y\ ;\ y := y+1\ \langle\!\langle q \rangle\!\rangle$, define
$r \equiv (now = \infty \wedge c?\ \textbf{during}\ [7,\infty)) \vee$
$\quad (\exists t \in [7,\infty) : now = t + T_{comm} + T_a \wedge (c, y)\ \textbf{at}\ t)$.
Note that we can derive $\langle\!\langle p \rangle\!\rangle\ c?y\ \langle\!\langle r \rangle\!\rangle$ and, using the nontermination axiom 6.4 and the disjunction rule, $\langle\!\langle r \rangle\!\rangle\ y := y+1\ \langle\!\langle q \rangle\!\rangle$. Hence the sequential composition rule leads to $\langle\!\langle p \rangle\!\rangle\ c?y\ ;\ y := y+1\ \langle\!\langle q \rangle\!\rangle$.

Rule 6.7. (Choice)
$$\frac{\langle\!\langle p \wedge b \rangle\!\rangle\ S_1\ \langle\!\langle q \rangle\!\rangle,\quad \langle\!\langle p \wedge \neg b \rangle\!\rangle\ S_2\ \langle\!\langle q \rangle\!\rangle}{\langle\!\langle p \rangle\!\rangle\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \langle\!\langle q \rangle\!\rangle}$$

The rule for the select construct **sel** $c?x$ **then** S_1 **or delay** e **then** S_2 **les** expresses that there are two possibilities. Either a communication along c occurs within e time units after the starting time t_0, leading to assertion p_1, after which S_1 is executed leading to q_1. Or the statement is waiting to communicate on c during e time units, i.e. no message is available, leading to assertion p_2, and S_2 is executed leading to q_2.

Rule 6.8. (Select)
$$\frac{\begin{array}{l}(p \wedge now < \infty)[t_0/now] \wedge \exists t \in [t_0, t_0+e) : comm(c,v)(t_0,t) \wedge \\ \quad now = t + T_{comm} \rightarrow p_1[v/x] \\ (p \wedge now < \infty)[t_0/now] \wedge c?\ \textbf{during}\ [t_0, t_0+e) \wedge now = t_0 + max(0,e) \rightarrow p_2 \\ \langle\!\langle p_i \rangle\!\rangle\ S_i\ \langle\!\langle q_i \rangle\!\rangle, \text{for } i = 1,2\end{array}}{\langle\!\langle p \wedge now < \infty \rangle\!\rangle\ \textbf{sel}\ c?x\ \textbf{then}\ S_1\ \textbf{or delay}\ e\ \textbf{then}\ S_2\ \textbf{les}\ \langle\!\langle q_1 \vee q_2 \rangle\!\rangle}$$

The rule for the while construct is related to the classical rule of Hoare Logic, but has a few extra clauses to deal with nonterminating computations. Further a delay statement has been included to model the time T_w representing the duration of the evaluation of the boolean.

Rule 6.9. (While)
$$\frac{\begin{array}{l}\langle\!\langle I \wedge b \wedge now < \infty \rangle\!\rangle\ \textbf{delay}\ T_w\ ;\ S\ \langle\!\langle I \rangle\!\rangle \\ \langle\!\langle I \wedge \neg b \wedge now < \infty \rangle\!\rangle\ \textbf{delay}\ T_w\ \langle\!\langle q \rangle\!\rangle \\ I \rightarrow I_0, \quad loc(I_0) = \emptyset \\ (\forall t_1 < \infty\ \exists t_2 > t_1 : I_0[t_2/now]) \rightarrow q_{nt}\end{array}}{\langle\!\langle I \rangle\!\rangle\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \langle\!\langle (q_{nt} \wedge now = \infty) \vee q \rangle\!\rangle}$$

We give a brief, informal, description of the soundness proof of this rule. For a model in the semantics of **while** b **do** S **od**, starting in a state satisfying I, there are four possibilities:

- The initial model, which satisfies I, is nonterminating, i.e., has a state σ_0 with $\sigma_0(now) = \infty$. Then a model of **while** b **do** S **od** by definition equals this model (this property is represented by the Nontermination Axiom). Since $now = \infty$ and $I \rightarrow I_0$ hold in this model, it satisfies $\forall t_1 < \infty\ \exists t_2 > t_1 : I_0[t_2/now]$, and then the third condition leads to q_{nt}. Thus the model satisfies $q_{nt} \wedge now = \infty$.
- It represents a terminating computation, obtained from a finite number of terminating computations of S. For all these computations of S, except for

the last one, b is true initially. By the first condition of the rule we can then prove by induction that I is true in the initial state of all these computations, again except for the last one. Since for the last computation $\neg b$ must be true, the second condition of the rule then leads to q for this model.

- It represents a nonterminating computation obtained from a nonterminating computation of S. Then, as in the previous case, we have $I \wedge b$ in the initial state of this last computation. Thus, using the first condition and the fact that it is a nonterminating computation, $I \wedge now = \infty$ holds for this model. Hence, since $I \rightarrow I_0$, we obtain $\forall t_1 < \infty\, \exists t_2 > t_1 : I_0[t_2/now]$, and then the third condition leads to q_{nt}.
- It represents a nonterminating computation obtained from an infinite sequence of terminating computations of S. Then, similar to the second case, the first condition leads by induction to the validity of I for all these computations. Since each computation of S takes at least T_w time units, which has a fixed positive lower bound (see Section 6.2), we have I, and thus I_0, after any point of time. Hence, $\forall t_1 < \infty\, \exists t_2 > t_1 : I_0[t_2/now]$, and then by third condition we obtain q_{nt}.

Example 6.6. We demonstrate the iteration rule by a small example which shows that our proof system is not restricted to partial correctness, but includes properties of both terminating and nonterminating computations. Consider

while $x \neq 0$ **do** $in?x$; $out!!f(x)$ **od**

Clearly this programs maintains a relation between input and output, represented by $\forall t < \infty\, \forall v : (in, v)$ **at** $t \rightarrow (out!!, f(v))$ **at** $(t + T_{comm})$. We do not prove this here (see Section 7 for such a proof), but concentrate in this example on the cases of termination and nontermination. We prove

$\langle\langle now = 0 \wedge x \neq 0 \rangle\rangle$

while $x \neq 0$ **do** $in?x$; $out!!f(x)$ **od**

$\langle\langle\, (now = \infty \wedge \exists t < \infty : in?$ **during** $[t, \infty)) \vee (now = \infty \wedge \forall t < \infty : \neg(in, 0)$ **at** $t) \vee$
$(now < \infty \wedge \exists t < \infty : (in, 0)$ **at** $t)\rangle\rangle$.

We use the iteration rule with

$q_{nt} \equiv (\exists t < \infty : in?$ **during** $[t, \infty)) \vee (\forall t < \infty : \neg(in, 0)$ **at** $t)$
$q \equiv now < \infty \wedge \exists t < \infty : (in, 0)$ **at** t
$I \equiv (now = \infty \wedge \exists t < \infty : in?$ **during** $[t, \infty)) \vee$
$(now < \infty \wedge \forall t < now, t \neq now - 2T_{comm} : \neg(in, 0)$ **at** $t \wedge$
$(x = 0 \leftrightarrow (in, 0)$ **at** $now - 2T_{comm}))$
$I_0 \equiv (\exists t < \infty : in?$ **during** $[t, \infty)) \vee (\forall t < now - 2T_{comm} : \neg(in, 0)$ **at** $t)$

To apply the iteration rule, we have to prove the following.

- $\langle\langle I \wedge x \neq 0 \wedge now < \infty \rangle\rangle$ **delay** T_w ; $in?x$; $out!!f(x)$ $\langle\langle I \rangle\rangle$.
The derivation of this formula by means of the proof system is rather straightforward. (Note that $I \wedge x \neq 0 \wedge now < \infty$ implies $\forall t < now : \neg(in, 0)$ **at** t.)
- $\langle\langle I \wedge x = 0 \wedge now < \infty \rangle\rangle$ **delay** T_w $\langle\langle q \rangle\rangle$.
Observe that $I \wedge x = 0 \wedge now < \infty$ implies $(in, 0)$ **at** $now - 2T_{comm}$, and hence $now < \infty \wedge \exists t < \infty : (in, 0)$ **at** t, i.e., q. Finally note that $\langle\langle q \rangle\rangle$ **delay** T_w $\langle\langle q \rangle\rangle$ can be derived easily.
- $I \rightarrow I_0$, which holds trivially. Further note that $loc(I_0) = \emptyset$.
- $(\forall t_1 < \infty\, \exists t_2 > t_1 : I_0[t_2/now]) \rightarrow q_{nt}$.
Observe that $\forall t_1 < \infty\, \exists t_2 > t_1 : I_0[t_2/now]$ is equivalent to $\forall t_1 < \infty\, \exists t_2 > t_1 :$

$(\exists t < \infty : in?$ **during** $[t, \infty)) \vee (\forall t < t_2 - 2T_{comm} : \neg(in, 0)$ **at** $t)$, which implies $(\exists t < \infty : in?$ **during** $[t, \infty)) \vee (\forall t < \infty : \neg(in, 0)$ **at** $t)$, i.e., q_{nt}.

Then the iteration rule leads to
$\langle\langle I \rangle\rangle$ **while** $x \neq 0$ **do** $in?x$; $out!!f(x)$ **od** $\langle\langle (q_{nt} \wedge now = \infty) \vee q \rangle\rangle$.
Note that $now = 0 \wedge x \neq 0 \rightarrow I$. Further, $(q_{nt} \wedge now = \infty) \vee q$ is equivalent to
$((\exists t < \infty : in?$ **during** $[t, \infty \vee \forall t < \infty : \neg(in, 0)$ **at** $t) \wedge now = \infty) \vee$
$(now < \infty \wedge \exists t < \infty : (in, 0)$ **at** $t)$ which implies
$(now = \infty \wedge \exists t < \infty : in?$ **during** $[t, \infty)) \vee (now = \infty \wedge \forall t < \infty : \neg(in, 0)$ **at** $t) \vee$
$(now < \infty \wedge \exists t < \infty : (in, 0)$ **at** $t)$.
Hence the consequence rule leads to the triple to be proved.

7. Example Chemical Batch – Final Implementation

In this section we implement component *Contr*, as specified in section 5:

$\langle\langle now = 0 \rangle\rangle$ *Contr* $\langle\langle CC \rangle\rangle$

where $CC \equiv CC_1 \wedge CC_2 \wedge CFC_1$ with

$CC_1 \equiv maxsend(\mathsf{actchan}, \mathsf{Init}, \mathsf{Period})$ **during** $[0, \infty)$

$CC_2 \equiv \forall t < \infty \, \forall v : (\mathsf{thermchan}, v)$ **at** $t \wedge v > \mathsf{SafeTemp} \rightarrow$
actchan!! **in** $[t, t + \mathsf{DelContr}]$

$CFC_1 \equiv \forall t < \infty : await$ thermchan? **in** $[t, t + \mathsf{DelReadTherm})$

We show that process *Contr* can be implemented by the program:

while *true* **do** thermchan?x ; **if** $x > \mathsf{SafeTemp}$ **then** actchan!!0 **fi od**

(Note that the value which is transmitted along actchan is irrelevant.)
Let $S \equiv$ thermchan?x ; **if** $x > \mathsf{SafeTemp}$ **then** actchan!!0 **fi**.
To prove $\langle\langle now = 0 \rangle\rangle$ **while** *true* **do** S **od** $\langle\langle CC \rangle\rangle$ we use the iteration rule with the following assertions:

$I_1 \equiv maxsend(\mathsf{actchan}, \mathsf{Init}, \mathsf{Period})$ **during** $[0, now - T_{comm}) \wedge$
$(\neg\mathsf{actchan}!!)$ **during** $(now - T_{comm}, now)$

$I_2 \equiv \forall t < now \, \forall v : (\mathsf{thermchan}, v)$ **at** $t \wedge v > \mathsf{SafeTemp} \rightarrow$
actchan!! **in** $[t, t + \mathsf{DelContr}]$

$I_3 \equiv \forall t < now - 2T_{comm} : await$ thermchan? **in** $[t, t + \mathsf{DelReadTherm})$

(Note that by the conventional properties of ∞ we have, e.g., $\infty - T_{comm} = \infty$. Thus, for instance, $I_1 \wedge now = \infty$ is equivalent to CC_1.)

Then define

$I \equiv I_1 \wedge I_2 \wedge I_3$ and

$I_0 \equiv I$.

Observe that $loc(I_0) = \emptyset$ and $\langle\langle I \wedge false \wedge now < \infty \rangle\rangle$ **delay** T_w $\langle\langle false \rangle\rangle$.

Further $\forall t_1 < \infty \, \exists t_2 > t_1 : I_0[t_2/now]$ implies $\forall t_1 < \infty \, \exists t_2 > t_1 :$
$maxsend(\mathsf{actchan}, \mathsf{Init}, \mathsf{Period})$ **during** $[0, t_2 - T_{comm}) \wedge$
$\forall t < t_2 \, \forall v : (\mathsf{thermchan}, v)$ **at** $t \wedge v > \mathsf{SafeTemp} \rightarrow$ actchan!! **in** $[t, t + \mathsf{DelContr}) \wedge$

$\forall t < t_2 - 2T_{comm}$: *await* thermchan? **in** $[t, t + \text{DelReadTherm})$ and hence $CC_1 \wedge CC_2 \wedge CFC_1$, that is, CC.

Then, assuming $\langle\langle I \wedge now < \infty\rangle\rangle$ **delay** T_w ; S $\langle\langle I\rangle\rangle$, the while rule leads to

$$\langle\langle I\rangle\rangle \textbf{ while } \textit{true} \textbf{ do } S \textbf{ od } \langle\langle CC \wedge now = \infty\rangle\rangle.$$

Since $now = 0 \rightarrow I$, the consequence rule leads to

$$\langle\langle now = 0\rangle\rangle \textbf{ while } \textit{true} \textbf{ do } S \textbf{ od } \langle\langle CC\rangle\rangle.$$

Hence it remains to prove $\langle\langle I \wedge now < \infty\rangle\rangle$ **delay** T_w ; S $\langle\langle I\rangle\rangle$. By the conjunction rule, this can be split up into the proof of

$$\langle\langle I_i \wedge now < \infty\rangle\rangle \textbf{ delay } T_w \text{ ; } S \ \langle\langle I_i\rangle\rangle, \text{ for } i = 1, 2, 3.$$

In the proof of I_i we give intermediate assertions p_i and r_i such that

$$\langle\langle I_i \wedge now < \infty\rangle\rangle \textbf{ delay } T_w \langle\langle p_i\rangle\rangle \tag{1i}$$

$$\langle\langle p_i\rangle\rangle \text{ thermchan?}x \ \langle\langle r_i\rangle\rangle \tag{2i}$$

$$\langle\langle r_i\rangle\rangle \textbf{ if } x > \text{SafeTemp} \textbf{ then } \text{actchan!!0} \textbf{ fi } \langle\langle I_i\rangle\rangle. \tag{3i}$$

For $I_1 \equiv maxsend(\text{actchan}, \text{Init}, \text{Period})$ **during** $[0, now - T_{comm})\wedge$
$(\neg\text{actchan!!})$ **during** $(now - T_{comm}, now)$
define

$p_1 \equiv now \geq T_w \wedge$ $(\neg\text{actchan!!})$ **during** $(now - T_{comm} - T_w, now)\wedge$
$maxsend(\text{actchan}, \text{Init}, \text{Period})$ **during** $[0, now - T_{comm} - T_w)$
$r_1 \equiv (I_1 \wedge now = \infty)\vee$
$(T_w + T_{comm} \leq now < \infty \wedge$ $(\neg\text{actchan!!})$ **during** $(now - 2T_{comm} - T_w, now)\wedge$
$maxsend(\text{actchan}, \text{Init}, \text{Period})$ **during** $[0, now))$

Then (11), (21), and (31) can be derived with

$\text{Init} = T_w + T_{comm}$,
$\text{Period} = T_w + 2T_{comm}$.

For $I_2 \equiv \forall t < now\ \forall v$: $(\text{thermchan}, v)$ **at** $t \wedge v > \text{SafeTemp} \rightarrow$
actchan!! **in** $[t, t + \text{DelContr}]$
we define

$p_2 \equiv I_2 \wedge now < \infty$
$r_2 \equiv (I_2 \wedge now = \infty)\vee$
$(now < \infty \wedge (\text{thermchan}, x)$ **at** $(now - T_{comm})\wedge$
$\forall t < now, t \neq now - T_{comm}\ \forall v$: $(\text{thermchan}, v)$ **at** $t \wedge v > \text{SafeTemp} \rightarrow$
actchan!! **in** $[t, t + \text{DelContr}])$

Then we can deduce (12) and (22), and prove (32) with

$\text{DelContr} = T_{comm}$.

For $I_3 \equiv \forall t < now - 2T_{comm}$: *await* thermchan? **in** $[t, t + \text{DelReadTherm})$
define

$p_3 \equiv now < \infty\wedge$
$\forall t < now - 2T_{comm} - T_w$: *await* thermchan? **in** $[t, t + \text{DelReadTherm})$
$r_3 \equiv (I_3 \wedge now = \infty)\vee$
$(now < \infty \wedge \forall t < now - T_{comm}$: *await* thermchan? **in** $[t, t + \text{DelReadTherm}))$.

This leads to (13) and (33), and we can derive (33), with
$\mathsf{DelReadTherm} = 2T_{comm} + T_w$.

7.1. Conclusion – Chemical Batch Processing

Recall that the refinement steps of section 5 have been proved correct, provided $\mathsf{MaxRise} > 0$ and $\mathsf{ExpTemp} - (\mathsf{SafeTemp} + \mathsf{ThermDev}) \geq \mathsf{MaxRise} \times (\mathsf{DelTherm} + \mathsf{DelReadTherm} + \mathsf{DelAct} + \mathsf{DelContr})$. Combining this with the constraints of this section, we have derived a correct system provided:

- The vessel, the thermometer, and the actuator satisfy their specification with $\mathsf{Init} = T_w + T_{comm}$, $\mathsf{Period} = T_w + 2T_{comm}$, and $\mathsf{MaxRise} > 0$.
- The parameter $\mathsf{SafeTemp}$ of the program satisfies $\mathsf{SafeTemp} \leq \mathsf{ExpTemp} - \mathsf{ThermDev} - \mathsf{MaxRise} \times (\mathsf{DelTherm} + 3T_{comm} + T_w + \mathsf{DelAct})$. Thus $\mathsf{SafeTemp}$ should be sufficiently smaller than the explosion temperature to cope with delays in the thermometer, the program, and the actuator, and the maximal rise of the temperature.

8. Concluding Remarks

To design distributed real-time systems, a compositional method based on Hoare logic has been presented. In this paper the maximal parallelism assumption has been used to model the situation that each process has its own processor. In [Hoo91] we describe a generalization to multiprogramming where several processes may share a single processor and scheduling is based on priorities. Further, [Hoo91] considers communication via synchronous channels where the sender waits until a communication partner is available. A combination of both types of channels and the use of local clocks in our framework is given in [Hoo93a].

To investigate the flexibility of our approach, in [Hoo93b, Hoo94] an application with a different communication mechanism is considered. There we specify and verify a distributed real-time arbitration protocol in which concurrent modules communicate by means of a common bus.

Since errors in real-time systems often have disastrous consequences, the reliability of such systems is often increased by techniques that should ensure the correct functioning of the system despite failures in some of its components. In general, fault-tolerance is achieved by some kind of redundancy which, however, takes time and therefore influences the timing behaviour of the system. Given this strong relation between real-time and fault-tolerance, we investigate the extension of our real-time framework to deal with fault-tolerance. To this end an atomic broadcast protocol has been specified and verified [ZhH95].

In this paper the framework has been illustrated by a chemical batch processing system. This example is taken from [ALF93], where a method is presented to obtain a requirements specification and to perform systematic analysis of safety requirements. Different formal techniques are applied during different stages of the analysis, leading to a specification of a safety controller by means of a Petri Net. Related to our approach is the use of modularity for the top-down developement of requirements.

This chemical batch processing example is a typical hybrid system, and a large number of methods to deal with such systems can be found in [GNR93]. Related to the first steps in our top-down approach is the Duration Calculus [CHR91], a real-time interval logic based on the concept of the duration of states. An important contribution of this calculus is the introduction of an integral primitive, expressing the duration of a certain state. Further we mention Lamport's temporal logic of actions (TLA) [AbL92], an assertional method which allows the top-down refinement of closed systems. It has been extended to real-time by adding a special variable *now* to represent time [Lam93].

Related to our formal framework is an early paper by Haase [Haa81] in which real-time is introduced as a variable in the data space of the program and assertions are derived by Dijkstra's weakest precondition calculus [Dij76]. A non-compositional approach can be found in [SBM92] where a logic of proof outlines with control predicates is extended to concurrent real-time programs by adding a primitive to express the time a control predicate last became true. A similar extension of Hoare logic is given in [Sha93] using a more general primitive to express the time that has elapsed since an assertion last held. Interesting in this last work is the use of the interactive proof checker PVS (Prototype Verification System) [ORS92]. Clearly for any system of reasonable size some tool support is indispensable. Therefore we are also experimenting with PVS to discharge simple verification conditions automatically and to check the proofs of design steps. Parts of the chemical batch processing example described here have already been specified and verified by means of PVS.

References

[Ada83] *The Programming Language Ada, Reference Manual*, 1983.

[ALF93] Anderson, T., de Lemos, R., Fitzgerald, J.S. and Saeed, A.: On formal support for industrial-scale requirements analysis. In *Workshop on Theory of Hybrid Systems*, pp. 426–451. LNCS 736, 1993.

[AbL92] Abadi, M. and Lamport, L.: An old-fashioned recipe for real-time. In *REX Workshop on Real-Time: Theory in Practice*, pp. 1–27. LNCS 600, Springer-Verlag, 1992.

[CHR91] Chaochen, Zhou, Hoare, C.A.R. and Ravn, A.P.: A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.

[Dij76] Dijkstra, E.W.: *A Discipline of Programming*. Prentice-Hall, 1976.

[GNR93] Grossman, R., Nerode, A., Ravn, A. and Rischel, H.: editors. *Hybrid Systems*. LNCS 736. Springer-Verlag, 1993.

[Haa81] Haase, V.H.: Real-time behaviour of programs. *IEEE Transactions on Software Engineering*, SE-7(5):494–501, 1981.

[Hoa69] Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[Hoo91] Hooman, J.: *Specification and Compositional Verification of Real-Time Systems*. LNCS 558, Springer-Verlag, 1991.

[Hoo93a] Hooman, J.: A compositional approach to the design of hybrid systems. In *Workshop on Theory of Hybrid Systems*, pp. 121–148. LNCS 736, 1993.

[Hoo93b] Hooman, J.: Specification and verification of a distributed real-time arbitration protocol. In *Proceedings 14th IEEE Real-Time Systems Symposium*, pp. 284–293. IEEE, 1993.

[Hoo94] Hooman, J.: Compositional verification of a distributed real-time arbitration protocol. *Real-Time Systems*, 6:173–205, 1994.

[HaP85] Harel, D. and Pnueli, A.: On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pp. 477–498. NATO, ASI-13, Springer-Verlag, 1985.

[Lam83] Lamport, L.: *What Good is Temporal Logic*, pp. 657–668. Information Processing, R.E. Manson (ed). North Holland, 1983.

[Lam93] Lamport, L.: Hybrid systems in TLA$^+$. In *Workshop on Theory of Hybrid Systems*, pp. 77–102. LNCS 736, 1993.

[Occ88] INMOS Limited. OCCAM *2 Reference Manual*, 1988.

[ORS92] Owre, S., Rushby, J. and Shankar, N.: PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pp. 748–752. Springer-Verlag, 1992.

[SBM92] Schneider, F., Bloom, B. and Marzullo, K.: Putting time into proof outlines. In *Workshop on Real-Time: Theory in Practice*, pp. 618–639. LNCS 600, Springer-Verlag, 1992.

[Sha93] Shankar, N.: Verification of real-time systems using PVS. In *Computer Aided Verification '93*, pp. 280–291. LNCS 697, Springer-Verlag, 1993.

[ZhH95] Zhou, P. and Hooman, J.: Formal specification and compositional verification of an atomic broadcast protocol. *Real-Time Systems*, to appear, 1995.

Received July 1993
Accepted in revised form June 1994