An introduction to compositional methods for concurrency and their application to real-time

Citation for published version (APA):

Hooman, J. J. M., & Roever, de, W. P. (1992). An introduction to compositional methods for concurrency and their application to real-time. *Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)*, 17(1), 29-73. <https://doi.org/10.1007/BF02811338>

DOI:

[10.1007/BF02811338](https://doi.org/10.1007/BF02811338)

Document status and date:

Published: 01/01/1992

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

An introduction to compositional methods for concurrency and their application to real-time

J J M HOOMAN¹ and W P de ROEVER²

¹Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

²Institute für Informatik and Praktische Mathematik II, Christian-Albrechts-Universität zu Kiel, Preusserstrasse 1–9, 2300 Kiel 1, Germany

Abstract. Formal methods to specify and verify concurrent programs with synchronous message passing are discussed. We stress the development towards compositional methods, i.e. methods in which the specification of a compound program can be inferred from specifications of its constituents without reference to the internal structure of those parts. Compositionality enables verification during the process of (top-down) design – the derivation of correct programs – instead of the more familiar a-posteriori verification based on already completed program codes. We sketch the transition from non-compositional towards compositional methods for concurrent programs, indicating the main principles behind compositionality. Having achieved a compositional framework based on classical Hoare triples, we discuss extensions to achieve a convenient formalism to specify and verify reactive systems that have an intensive interaction with their environment. Next this Hoare-style framework is adapted to specify and verify real-time properties, and a compositional proof method is formulated for real-time distributed computing. Compositional reasoning during top-down development of a real-time program is illustrated by an example concerning a watchdog timer.

Keywords. Compositional methods for concurrency; real-time applications; distributed computing.

1. Introduction

Formal methods for the specification and verification of distributed systems can be classified from the viewpoint of expressibility (which properties can be specified), specification language (e.g., temporal logic, Hoare triples and first-order assertions), and programming features (such as time-out, various communication mechanisms and concurrency). In this paper we concentrate on the distinction between proof methods that are only applicable to complete program code and methods that can be used to verify design steps during the process of program development. We sketch

the development from a-posteriori methods (requiring the complete program text) towards compositional methods (supporting verify-while-design). Compositionality can be considered as a requirement for hierarchical, structured, program derivation. A separation of concerns is desired between the use of (and the reasoning about) a module and its implementation. This leads to the following definition of *compositionality* for proof methods:

Properties of a compound programming language construct (such as sequential composition and parallel composition) can be deduced from specifications for its constituent parts without any further information about the internal structure of these parts.

In general, compositional program specification and verification dictates, as a principle, that all aspects of program execution which are required to define the meaning of a compound statement from its constituents, must be explicitly addressed in semantics and assertion language alike. In *semantics* because, otherwise, no compositional semantics can be defined, since compositionality in semantics requires that the meaning of a compound statement is a function of the meaning of its parts (the guiding principle of denotational semantics). In *specification languages* because, otherwise, no compositional verification rules can be formulated in which the specification of a compound statement should follow from specifications of its constituent parts without knowledge about their internal structure (the internal structure often providing implicit information which has not been explicitly stated in the specification, but is used in non-compositional methods (Owicki & Gries 1976; Apt *et al* 1980)). The rationale for this principle is that one must be able to specify the behaviour of a module in isolation, i.e. without any implicit prior assumption regarding the environment within which it ultimately functions. Hence, all assumptions which are needed regarding the environment—because these influence the behaviour of a module—must be made explicit as parameters (in the semantics and specification of that module alike) for only then one can abstract away from the remaining aspects (such as inner syntactic structure).

In case of shared variable communication this compositionality principle implies that when defining the behaviour of a module any change of a shared variable by the environment must be explicitly expressed as an assumption of that module regarding its environment. This is worked out in Aczel's model for shared variable semantics as cited in de Roever (1985b, pp. 181–207). Similarly, when considering distributed communication via input/output-statements, the specification of, e.g., an input statement in one module requires explicit expressibility of assumptions regarding a corresponding output statement in another module. In case one abstracts away from blocking behaviour only assumptions regarding the value communicated must be expressible. If blocking behaviour is a focus of interest, this is again an assumption regarding program execution which must be stated explicitly; i.e. one has to state the effect of no communication partner being available in the assertion language and one must be able to express the assumption that no partner is available in the assertion language.

In this paper we also discuss the compositional verification of real-time properties for distributed systems. When the timing behaviour of a statement is considered, all factors concerning the execution of this statement which influence that timing behaviour must be expressible. For example, for real-time systems we use in this paper the maximal progress assumption with respect to distributed i/o-communication

from Koymans *et al* (1988): no input or output statement should wait for communication when its partner is also ready to communicate. This aspect of timing behaviour requires, indeed, that one must be able to express when a partner is waiting to communicate. For, otherwise, maximal progress would not be expressible within the semantics, and hence timing behaviour of i/o-statements could not be characterized. This maximal progress assumption, which represents the situation that each process has its own processor, can be generalized to multiprogramming where several processes may share a single processor. By introducing priorities for processes on a single processor, certain statements which are ready to execute will not be executed on account of their priority and because at most one action can be executed at a time on a uniprocessor. Modelling the timing behaviour of such statements requires that the semantics, and hence the specification language, contains primitives to state explicitly when a statement is executing and when it is requesting processor time with a certain priority. The semantic aspects of reasoning formally about real-time and scheduling by means of priorities are addressed technically in Hooman (1991a).

This paper is structured as follows. A programming language with synchronous message passing is defined in §2. Section 3 contains a description of a classical non-compositional method, and we indicate how a compositional proof system can be achieved for Hoare triples (pre-condition, program, post-condition). For the specification and verification of reactive systems, these triples are extended in §4 with assertions (called assumption and commitment) that specify the communication interface between a program and its environment. In §5 we adapt this Hoare-style framework to specify real-time properties of programs, and we give the details of a compositional proof system for real-time distributed systems. The formalism of §5 is illustrated by an example of a watchdog timer in §6. The extension of this formalism to assumption/commitment based reasoning for real-time is described in §7. In §8 we sketch the development of the field, leading to a description of the state of the art and the place of our work therein.

2. Syntax

We give syntax and informal semantics of a programming language for distributed synchronous message-passing. Our language is akin to Occam (1988) with concurrent processes that communicate via message passing along unidirectional channels, each connecting two processes. Communication is synchronous, i.e., both the sender and the receiver have to wait until a communication partner is available.

Let $CHAN$ be a nonempty set of channel names, VAR be a nonempty set of program variables, and VAL be a denumerable domain of values. $\mathbb{N}$ denotes the set of natural numbers (including 0). The syntax of our programming language is given in table 1, with $n \in \mathbb{N}$, $n \geq 1$, $c, c_1, \dots, c_n \in CHAN$, $x, x_1, \dots, x_n \in VAR$, and $\vartheta \in VAL$.

Table 1. Syntax programming language.

<i>Expression</i>	$e ::= \vartheta \mid x \mid e_1 + e_2 \mid e_1 + e_2 \mid e_1 \times e_2$
<i>Boolean expression</i>	$b ::= e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
<i>Statement</i>	$S ::= x := e \mid c!e \mid c?x \mid S_1; S_2 \mid G \mid \star G \mid S_1 \parallel S_2$
<i>Guarded command</i>	$G ::= [\square_{i=1}^n b_i \rightarrow S_i] \mid [\square_{i=1}^n b_i; c_i?x_i \rightarrow S_i]$

Informally, the statements of our programming language have the following meaning.

Atomic statements

- Assignment $x := e$ assigns the value of expression e to the variable x .
- Output statement $c!e$ is used to send the value of expression e on channel c as soon as an input command $c?x$ is available. Since we assume synchronous communication, such an output statement is suspended until a parallel process executes a corresponding input statement.
- Input statement $c?x$ is used to receive a value via channel c and assign this value to the variable x . As for the output command, such an input statement has to wait for a corresponding partner before a (synchronous) communication can take place.

Henceforth we will often refer to an input or output statement as an *i/o-statement*.

Compound statements

- $S_1; S_2$ indicates sequential composition: first execute S_1 , and continue with the execution of S_2 if and when S_1 terminates.
- Guarded command $[\bigvee_{i=1}^n b_i \rightarrow S_i]$. If none of the b_i evaluate to true then this guarded command terminates after evaluation of the booleans. Otherwise, non-deterministically select one of the b_i that evaluates to true and execute the corresponding statement S_i .
- Guarded command $[\bigvee_{i=1}^n b_i; c_i?x_i \rightarrow S_i]$. A guard (the part before the arrow) is *open* if its boolean part evaluates to true. If none of the guards is open, the guarded command terminates after evaluation of the booleans. Otherwise, wait until the communication of one of the open guards can be performed and continue with the corresponding S_i .
- Iteration $*G$ indicates repeated execution of guarded command G as long as at least one of the guards is open. When none of the guards is open $*G$ terminates.
- $S_1 \parallel S_2$ indicates parallel execution of the statements S_1 and S_2 . The components S_1 and S_2 of a parallel composition are often called *processes*.

Henceforth we use $\equiv$ to denote syntactic equality. Conventional abbreviations are used, such as $true \equiv 0 = 0$, $false \equiv \neg true$, $b_1 \wedge b_2 \equiv \neg(\neg b_1 \vee \neg b_2)$ etc.

For a guarded command $G \equiv [\bigvee_{i=1}^n b_i \rightarrow S_i]$ or $G \equiv [\bigvee_{i=1}^n b_i; c_i?x_i \rightarrow S_i]$, we define $b_G \equiv b_1 \vee \dots \vee b_n$. Observe that conventional programming constructs can be defined as an abbreviation:

$$\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \equiv [b \rightarrow S_1 \square \neg b \rightarrow S_2] \text{ and } \text{while } b \text{ do } S \text{ od} \equiv \star[b \rightarrow S].$$

3. Compositionality

In §3.1 we explain the principles of traditional non-compositional methods. The development towards compositional proof systems based on Hoare triples is described in §3.2.

3.1 Non-compositional methods

Classical verification methods for parallel processes, such as Owicki & Gries (1976) for shared variable communication and Apt *et al* (1980), Levin & Gries (1981) for synchronous message passing, consist of two stages. First a *local* correctness proof is

given for each of the sequential process by associating assertions with locations in the program. In the second, *global*, stage a consistency check is applied to the local proofs.

- For shared variables this is the *interference freedom* test which verifies that assertions in the proof of one process remain valid under actions of other processes.
- For communication via message passing the *cooperation* test is applied to verify correctness of assertions attached to locations after input- and output-statements.

Such methods are not compositional because at parallel composition they require the complete program text, annotated with assertions, of the constituent processes. Moreover, they are only suited for top-level parallelism, that is, to prove correctness of programs of the form $S_1 \parallel \dots \parallel S_n$ where $S_1, \dots, S_n$ are sequential processes.

As an example, we consider in more detail the method of Apt *et al* (1980) for synchronous message passing. This method is based on *Hoare triples* (Hoare 1969) that is, on correctness formulae of the form $\{p\} S \{q\}$ which have the following meaning: if we start program S in a state satisfying assertion p (the pre-condition) and if program S terminates then assertion q (the post-condition) holds for the termination state. For example, $\{x = 5\} x := x + 1 \{x = 6\}$ is a valid Hoare triple.

First we indicate how a proof system can be formulated in which valid Hoare triples can be derived for sequential programs. Let $q[e/x]$ denote the textual substitution of expression e for each free occurrence of variable x in assertion q . Then we have the following assignment axiom:

Axiom 3.1. (Assignment)

$$\{q[e/x]\} x := e \{q\}.$$

Example 3.1. With this axiom we can derive $\{x = 5\} x := x + 1 \{x = 6\}$, because $(x = 6)[x + 1/x]$ equals $x + 1 = 6$, which is equivalent to $x = 5$.

Furthermore the proof system contains rules for compound constructs. For instance, sequential composition is modelled by the following rule:

Rule 3.2. (Sequential composition)

$$\frac{\{p\} S_1 \{r\}, \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}}.$$

By such a rule the formula below the line can be derived from the formulae above the line. Soundness of the rule is proved by showing that validity of the formulae above the line implies that the formula below the line is valid. Note that this rule is compositional because the formula for $S_1; S_2$ is derived without using the structure of S_1 or S_2 . To strengthen pre-conditions and weaken post-conditions, the proof system contains the following rule:

Rule 3.3. (Consequence)

$$\frac{p \rightarrow p_1, \{q_1\} S \{q_1\}, q_1 \rightarrow q}{\{p\} S \{q\}}.$$

To illustrate the rule for parallel composition in Apt *et al* (1980), we consider the proof of

$$\{y = 3\} \{a?x; x := x + 1; b!(x + 2)\} \parallel \{a!y; b?y; y := y + 2\} \{x = 4 \wedge y = 8\}.$$

In the first stage we attach assertions to all locations in the program text of the two processes, leading to so-called *proof outlines*:

$$\{\text{true}\} a?x \{x = 3\}; x := x + 1 \{x = 4\}; b!(x + 2) \{x = 4\},$$

and

$$\{y = 3\} a!y \{y = 3\}; b?y \{y = 6\}; y := y + 2 \{y = 8\}.$$

In this stage only the post-conditions of assignments are verified: from the assignment axiom we obtain $\{x = 3\} x := x + 1 \{x = 4\}$ and $\{y = 6\} y := y + 2 \{y = 8\}$. Observe that the post-conditions of the input statements $a?x$ and $b?y$ express assumptions about the values sent by the communication partner.

These assumptions are verified in the second stage by means of the *cooperation test*.¹ In general, this test requires that for $\{p_1\} c?x \{q_1\}$ and $\{p_2\} c!e \{q_2\}$ in the proof outlines of two processes we have to prove $\{p_1 \wedge p_2\} c?x \parallel c!e \{q_1 \wedge q_2\}$, which is equivalent to proving $\{p_1 \wedge p_2\} x := e \{q_1 \wedge q_2\}$. In our example this leads to the proof obligations:

$$\{\text{true} \wedge y = 3\} a?x \parallel a!y \{x = 3 \wedge y = 3\}$$

and

$$\{y = 3 \wedge x = 4\} b?y \parallel b!(x + 2) \{y = 6 \wedge x = 4\} \text{ which are easy to prove.}$$

After the verification of the first two stages we obtain the conjunction of all pre-conditions from the sequential processes as the pre-condition of the complete program and the conjunction of the post-conditions as the final post-condition. In our example this leads to the pre-condition $\text{true} \wedge y = 3$ and the post-condition $x = 4 \wedge y = 8$ which are equivalent to the required conditions.

3.2 Towards compositionality

In this section we discuss how a compositional proof method can be obtained for programs which communicate via synchronous message passing. First the cooperation test from Apt *et al* (1980) is removed by not allowing implicit assumptions in the post-conditions of i/o-statements. The local proof of a sequential program should be valid in any arbitrary environment. Since this would weaken the method, and any valid post-condition should be provable, we use a *history variable* h which denotes the communication history of the complete program. A (communication) *history* is a sequence of records $\langle c, \vartheta \rangle$ where c is a channel name and ϑ a value. For example, $\langle (c, 5), (b, 6), (a, 8), (b, 0) \rangle$ is a history expressing four communications: first one via channel c with value 5, then a communication via b with value 6, etc. Let $\langle \rangle$ denote the empty sequence. History variable h does not appear in the program, but it is updated implicitly in the semantics of i/o-statements. This leads to the following valid formulae.

- For an output command we have, for example,

$$\{h = \langle (c, 5) \rangle\} b!6 \{h = \langle (c, 5), (b, 6) \rangle\}.$$

¹ In the full method of Apt *et al* (1980) auxiliary variables and a global invariant are used to make the method complete, i.e., to guarantee that any valid Hoare triple can be proved.

- For an input statement $c?x$ we can only express in the post-condition that there exists a value which is communicated via c and assigned to x . For instance,

$$\{h = \langle \rangle\} c?x \{ \exists v: h = \langle (c, v) \rangle \wedge x = v \}.$$

Example 3.2. To prove $\{true\} c!5 \parallel c?x \{x = 5\}$, we use the Hoare triples

$$\{h = \langle \rangle\} c?5 \{h = \langle (c, 5) \rangle\} \text{ and } \{h = \langle \rangle\} c?x \{ \exists v: h = \langle (c, v) \rangle \wedge x = v \}.$$

Suppose the pre- and post-conditions after parallel composition are obtained by simply taking the conjunction of, respectively, pre- and post-conditions of the sequential programs. Then

$$\{h = \langle \rangle\} c!5 \parallel c?x \{h = \langle (c, 5) \rangle \wedge \exists v: h = \langle (c, v) \rangle \wedge x = v\}.$$

Since the post-condition implies $\exists v: v = 5 \wedge x = v$, and hence $x = 5$, the consequence rule leads to $\{h = \langle \rangle\} c!5 \parallel c?x \{x = 5\}$. By a so-called *substitution rule* (not given in this paper), we could substitute $\langle \rangle$ for h in the pre-condition, thus obtaining pre-condition *true*. $\square$

Example 3.2 suggests the following rule:

$$\frac{\{p_1\} S_1 \{q_1\}, \{p_2\} S_2 \{q_2\}}{\{p_1 \wedge p_2\} S_1 \parallel S_2 \{q_1 \wedge q_2\}}.$$

Example 3.3. Consider again $S_1 \parallel S_2$, where $S_1 \equiv a?x; x := x + 1; b!(x + 2)$ and $S_2 \equiv a!y; b?y; y := y + 2$. First derive the following Hoare triples:

$$\{h = \langle \rangle\} a?x \{ \exists v_1: h = \langle (a, v_1) \rangle \wedge x = v_1 \},$$

$$\{ \exists v_1: h = \langle (a, v_1) \rangle \wedge x = v_1 \} x := x + 1 \{ \exists v_1: h = \langle (a, v_1) \rangle \wedge x = v_1 + 1 \},$$

and

$$\{ \exists v_1: h = \langle (a, v_1) \rangle \wedge x = v_1 + 1 \} b!(x + 2)$$

$$\{ \exists v_1: h = \langle (a, v_1) \rangle, (b, v_1 + 3) \rangle \wedge x = v_1 + 1 \}.$$

By two applications of the sequential composition rule we obtain

$$\{h = \langle \rangle\} a?x; x := x + 1; b!(x + 2)$$

$$\{ \exists v_1: h = \langle (a, v_1) \rangle, (b, v_1 + 3) \rangle \wedge x = v_1 + 1 \}.$$

Similarly,

$$\{h = \langle \rangle \wedge y = 3\} a!y; b?y; y := y + 2 \{ \exists v_2: h = \langle (a, 3) \rangle, (b, v_2) \rangle \wedge y = v_2 + 2 \}.$$

Then the parallel composition rule above leads to

$$\begin{aligned} \{h = \langle \rangle \wedge y = 3\} S_1 \parallel S_2 \{ \exists v_1: h = \langle (a, v_1) \rangle, (b, v_1 + 3) \rangle \wedge x = v_1 + 1 \wedge \\ \exists v_2: h = \langle (a, 3) \rangle, (b, v_2) \rangle \wedge y = v_2 + 2 \}. \end{aligned}$$

The post-condition implies $\exists v_1, v_2: v_1 = 3 \wedge v_2 = v_1 + 3 \wedge x = v_1 + 1 \wedge y = v_2 + 2$, which leads to $x = 4 \wedge y = 8$. Thus, by the consequence rule. $\{h = \langle \rangle \wedge y = 3\} S_1 \parallel S_2 \{x = 4 \wedge y = 8\}$.

(Again $h = \langle \rangle$ in the pre-condition can be removed by a substitution rule.) $\square$

Although this works nicely for two processes, the next example shows that there is a problem if more than two processes are involved.

Example 3.4. Consider $S_1 \parallel S_2 \parallel S_3$, where $S_1 \equiv a!0; b?x$, $S_2 \equiv a?y; c!(y+1)$, and $S_3 \equiv c?z; b!(z+1)$. Similar to example 3.3, we could first prove

$$\{h = \langle \rangle\} S_1 \{q_1 \equiv \exists v_1 : h = \langle (a, 0), (b, v_1) \rangle \wedge x = v_1\}$$

and

$$\{h = \langle \rangle\} S_2 \{q_2 \equiv \exists v_2 : h = \langle (a, v_2), (c, v_2 + 1) \rangle \wedge y = v_2\}.$$

But then the conjunction of q_1 and q_2 implies *false* whereas $S_1 \parallel S_2 \parallel S_3$ terminates and hence does not satisfy post-condition *false*.

The problem is that h denotes the global history of the complete program – e.g., consisting of three processes – whereas each of the processes in isolation can only describe the history on its own channels. A possible solution is to give each process its own history variable, and to combine these local history variables at parallel composition. This is done in Soundararajan (1984a), using a predicate *compat*. Zwiers (Zwiers *et al* 1984; Zwiers 1989), however, shows that a concise and simple rule for parallel composition can be formulated if each process uses projections of global history variable h onto its own channels. Such a projection expresses the view of a particular process on the global history. Formally, the *projection* of h onto a set of channel names $cset$, notation h_{cset} , denotes the sequence obtained from the history denoted by h by removing all records with a channel name not in $cset$. For instance, if $h = \langle (a, 0), (c, 1), (b, 3) \rangle$ then $h_{\{c\}} = \langle (c, 1) \rangle$, $h_{\{b, c\}} = \langle (c, 1), (b, 3) \rangle$, and $h_{\{a\}} = \langle \rangle$. Henceforth we write h_c , h_{bc} , and h_a instead of $h_{\{c\}}$, $h_{\{b, c\}}$, and $h_{\{a\}}$, respectively.

In the rule for $S_1 \parallel S_2$ we require that the post-condition of S_i only refer to history h via projections on channels occurring in S_i , for $i = 1, 2$. If, moreover, the post-condition of S_i only refers to program variables of S_i , then the following rule for parallel composition is sound;

Rule 3.4. (Parallel composition)

$$\frac{\{p_1\} S_1 \{q_1\}, \{p_2\} S_2 \{q_2\}}{\{p_1 \wedge p_2\} S_1 \parallel S_2 \{q_1 \wedge q_2\}}.$$

Observe that this is a compositional rule because a Hoare triple for $S_1 \parallel S_2$ can be derived without knowing the internal structure of S_1 and S_2 . To obtain a valid rule we only impose a simple syntactic requirement on assertions and processes (the post-condition of a process should only refer to channels and variables of the process itself). This simple syntactic check replaces the cooperation test which requires proof outlines for the complete program text of the processes.

Except for bottom-up verification such a compositional rule can be used for top-down development. Therefore, a triple $\{p\} S \{q\}$ is considered as a specification for a program S . Suppose we decide to implement S as $S_1 \parallel S_2$. If we can find assertions p_1 , q_1 , p_2 , and q_2 that satisfy $p \rightarrow p_1 \wedge p_2$ and $q_1 \wedge q_2 \rightarrow q$, and moreover certain syntactic requirements on the post-conditions hold, then S_1 and S_2 can be implemented independently, using specifications $\{p_1\} S_1 \{q_1\}$ and $\{p_2\} S_2 \{q_2\}$.

Since in this compositional framework programs can be considered as black boxes, and verification is done on the basis of the specifications only, we can allow nested parallelism in programs as expressed by the syntax of § 2.

4. Extensions of Hoare triples

A Hoare triple is perfectly suited to describe the observable behaviour of a sequential program which is given by initial and final states. For a parallel program also the communication behaviour on its external channels is observable. Hence a specification of a parallel component should express this communication interface. Note, however, that a specification $\{p\}S\{q\}$ has an important limitation: it only specifies the behaviour of S if S terminates. All non-terminating computations of S satisfy such a specification trivially. Thus the post-condition cannot be used to express the communication interface of non-terminating programs. Therefore, a Hoare triple is extended with an invariant, called *commitment* in this paper, which must hold throughout the computation. This leads to a formula of the form $C: \{p\}S\{q\}$ where commitment C describes the communication interface of S during its execution. The success of such formulae in many applications is based on a simple rule for parallel composition in which, besides conjunctions for pre- and post-conditions, also the conjunction of commitments can be taken.

Rule 4.1. (Parallel composition)

$$\frac{C_1: \{p_1\}S_1\{q_1\}, C_2: \{p_2\}S_2\{q_2\}}{C_1 \wedge C_2: \{p_1 \wedge p_2\}S_1 \parallel S_2\{q_1 \wedge q_2\}}$$

provided, for $i = 1, 2$, the assertions C_i and q_i refer to h only via projections on the channels occurring in S_i , and q_i only refers to program variables of S_i .

In this formalism the influence of the environment on the communication behaviour of a program can be expressed by using implications in the commitment. The next example indicates that for so-called reactive systems (Harel & Pnueli 1985) which have an intensive interaction with their environment, this style of specification often leads to proofs using inductive arguments. This motivates a final extension of the Hoare-style formulae.

In the following examples, $seq_1 \leq seq_2$ expresses that sequence seq_1 is an initial prefix of sequence seq_2 . For a history variable h , a set of channels $cset$, and a number i , we use $h_{cset}[i] = (c, \vartheta)$ to denote that the record (c, ϑ) is the i th element of the sequence denoted by h_{cset} . We assume that $h_{cset}[i] = (c, \vartheta)$ is true if i is greater than the length of h_{cset} .

Example 4.1. We verify two reactive, non-terminating, processes that have a close interaction. Consider $S_1 \equiv c!1; \star[d?x \rightarrow c!(x+1)]$ and $S_2 \equiv \star[c?y \rightarrow d!(y+1)]$. The aim is to prove that $S_1 \parallel S_2$ satisfies the commitment $h_{cd} \leq \langle (c, 1), (d, 2), (c, 3), (d, 4), (c, 5), \dots \rangle$, that is,

$$\forall i \geq 1: h_{cd}[2i-1] = (c, 2i-1) \wedge h_{cd}[2i] = (d, 2i).$$

Define

$$C_1 \equiv h_{cd}[1] = (c, 1) \wedge (\forall i \geq 1 \forall v: h_{cd}[2i] = (d, v) \rightarrow h_{cd}[2i+1] = (c, v+1)).$$

Then for S_1 we can prove

$$C_1: \{h_{cd} = \langle \rangle\} c!1; \star[d?x \rightarrow c!(x+1)] \{false\}.$$

Similarly, for S_2 we define

$$C_2 \equiv (\forall i \geq 1 \forall v: h_{cd}[2i-1] = (c, v) \rightarrow h_{cd}[2i] = (d, v+1)).$$

Then we have

$$C_2: \{h_{cd} = \langle \rangle\} \star [c?y \rightarrow d!(y+1)] \{false\}.$$

Since the required syntactic conditions are fulfilled, we can apply the rule for parallel composition which leads to

$$C_1 \wedge C_2: \{h_{cd} = \langle \rangle\} S_1 \parallel S_2 \{false\}.$$

To prove that $C_1 \wedge C_2$ implies the required commitment, we prove by induction on i that, for $i \geq 1$, $h_{cd}[2i-1] = (c, 2i-1) \wedge h_{cd}[2i] = (d, 2i)$.

- Basic step. For $i = 1$ we have, by C_1 , that $h_{cd}[1] = (c, 1)$ and thus from C_2 we obtain $h_{cd}[2] = (d, 2)$.
- Induction step. Assume $h_{cd}[2i-1] = (c, 2i-1) \wedge h_{cd}[2i] = (d, 2i)$. Then $h_{cd}[2i] = (d, 2i)$ implies, by C_1 , that $h_{cd}[2i+1] = (c, 2i+1)$, and thus $h_{cd}[2(i+1)-1] = (c, 2(i+1)-1)$. Using C_2 this leads to $h_{cd}[2(i+1)] = (d, 2(i+1))$. $\square$

This example illustrates that assumptions about the environment are important in the specification of a process, and it indicates that mutual assumptions of processes about each others communication interface usually leads to correctness proofs using inductive reasoning. Based on these observations we present an extension of the correctness formulae in which a process can be specified relative to explicit assumptions about its environment, and an inductive relation is incorporated in the specification. Therefore the specification formula is extended with a second invariant, called *assumption*, which expresses assumptions about the environment and by which we can strengthen post-condition and commitment. This leads to formulae of the form $(A, C): \{p\} S \{q\}$, where

A is an *assumption* describing the expected behaviour of the environment of S , and C is a *commitment* which is guaranteed by process S itself, as long as the environment does not violate the assumption.

The general idea is that assumption and commitment reflect the communication interface between parallel components (and hence do not contain program variables), whereas pre- and post-condition facilitate the reasoning at sequential composition.

Example 4.2. In this formalism assumptions about the values sent by the environment can be expressed explicitly. For instance, the assertion $h_c \leq \langle (c, 3) \rangle$ can be used as an assumption:

$$(h_c \leq \langle (c, 3) \rangle, true): \{true\} c?x \{x = 3\}.$$

This assumption expresses that if a communication along c takes place then the environment will send the value 3. The next formula shows that it can be used for a commitment about the next communication:

$$(h_c \leq \langle (c, 3) \rangle, h_a \leq \langle (a, 4) \rangle): \{true\} c?x; a!(x+1) \{x = 3\}. \quad \square$$

Assumption/commitment based reasoning was introduced in Misra & Chandy (1981). A proof system for these formulae has been given in Zwiers *et al* (1984). In this paper we discuss the proof obligations for assumptions and commitments in the parallel composition rule. Consider the parallel composition $S_1 \parallel S_2$, and suppose we have assumption-commitment pairs (A_1, C_1) for S_1 and (A_2, C_2) for S_2 . Which

conditions have to be verified to obtain a pair (A, C) for $S_1 \parallel S_2$? Consider assumption A_2 of S_2 :

- A_2 may contain assumptions about joint channels of S_1 and S_2 which connect these two processes; these assumptions must be justified by the commitment C_1 of S_1 .
- A_2 may contain assumptions about external channels of S_2 . These assumptions are maintained in the new network assumption A for $S_1 \parallel S_2$.

This leads to the following proof obligation: $A \wedge C_1 \rightarrow A_2$. Similarly, $A \wedge C_2 \rightarrow A_1$.

To obtain a sound rule with these implications, the meaning of a formula $(A_i, C_i): \{p_i\} S_i \{q_i\}$ has to be defined carefully. A simple implication between A_i and C_i would with the implications above and $A \equiv \text{true}$ lead to circular reasoning, that is, $A_1 \rightarrow C_1 \rightarrow A_2 \rightarrow C_2 \rightarrow A_1$. Therefore in defining the meaning of $(A_i, C_i): \{p_i\} S_i \{q_i\}$ we require that if p_i holds in the initial state then (1) C_i holds initially, and (2) C_i holds after every communication provided A_i holds after all preceding communications. This inductive step inside the meaning of formulae is sufficient to avoid circularity (see Misra & Chandy 1981, Zwiers *et al* 1984).

As in rule 4.1 we can take the conjunction of pre-conditions, post-conditions, and commitments, provided, for $i = 1, 2$, the assertions A_i , C_i , p_i and q_i of S_i refer only to h via projections on the channels of S_i , and p_i and q_i only refer to program variables of S_i . (Program variables are not allowed in A_i and C_i .) With these constraints, the following rule for parallel composition is valid.

Rule 4.2. (Parallel composition A-C)

$$\frac{(A_1, C_1): \{p_1\} S_1 \{q_1\}, (A_2, C_2): \{p_2\} S_2 \{q_2\} \quad A \wedge C_1 \rightarrow A_2, A \wedge C_2 \rightarrow A_1}{(A, C_1 \wedge C_2): \{p_1 \wedge p_2\} S_1 \parallel S_2 \{q_1 \wedge q_2\}}$$

Example 4.3. Consider $S_1 \parallel S_2$ where $S_1 \equiv a?x; x = x + 1; b!(x + 2)$ and $S_2 \equiv c?y; a!y; b?y; y = y + 2$. Then for S_1 and S_2 we can derive

$$(A_1 \equiv h_a \leq \langle (a, 3) \rangle, C_1 \equiv h_b \leq \langle (b, 6) \rangle): \{h_{ab} = \langle \rangle\} S_1 \{x = 4\},$$

and

$$(A_2 \equiv h_c \leq \langle (c, 3) \rangle \wedge h_b \leq \langle (b, 6) \rangle, C_2 \equiv h_a \leq \langle (a, 3) \rangle): \{h_{abc} = \langle \rangle\} S_2 \{y = 8\}.$$

Since S_1 and S_2 communicate with each other via the channels a and b , the remaining assumption for $S_1 \parallel S_2$ concerns the external channel c : $A \equiv h_c \leq \langle (c, 3) \rangle$.

Then $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$, thus the parallel composition rule leads to

$$(A, C_1 \wedge C_2): \{h_{abc} = \langle \rangle\} S_1 \parallel S_2 \{x = 4 \wedge y = 8\}. \quad \square$$

Example 4.4. Consider again the two reactive processes from example 4.1:

$$S_1 \equiv c!1; \star[d?x \rightarrow c!(x + 1)] \text{ and } S_2 \equiv \star[c?y \rightarrow d!(y + 1)].$$

We show how the assumption/commitment formalism can be used to prove for $S_1 \parallel S_2$ the commitment $\forall i \geq 1: h_{cd}[2i - 1] = (c, 2i - 1) \wedge h_{cd}[2i] = (d, 2i)$.

Define

$$A_1 \equiv \forall i \geq 1: h_{cd}[2i] = (d, 2i)$$

and

$$C_1 \equiv \forall i \geq 1: h_{cd}[2i - 1] = (c, 2i - 1).$$

Then for S_1 we can prove

$$(A_1, C_1): \{h_{cd} = \langle \rangle\} c!1; \star [d?x \rightarrow c!(x+1)] \{false\}.$$

Similarly, for S_2 we define

$$A_2 \equiv \forall i \geq 1: h_{cd}[2i-1] = (c, 2i-1),$$

and

$$C_2 \equiv \forall i \geq 1: h_{cd}[2i] = (d, 2i).$$

Then we have

$$(A_2, C_2): \{h_{cd} = \langle \rangle\} \star [c?y \rightarrow d!(y+1)] \{false\}.$$

Let $A \equiv true$. Then $A \wedge C_2 \rightarrow A_1$ and $A \wedge C_1 \rightarrow A_2$. Since the other conditions on the assertions are also satisfied, we can apply the rule for parallel composition, leading to

$$(true, C_1 \wedge C_2): \{h_{cd} = \langle \rangle\} S_1 \parallel S_2 \{false\}.$$

Clearly $C_1 \wedge C_2$ is equivalent to the required commitment. $\square$

Observe that in the correctness proof for the two reactive processes from example 4.4 there is no explicit inductive argument. The requirement inductive reasoning is performed only once in the soundness proof of the parallel composition rule (rule 4.2). In this respect we have obtained a rule for parallel composition which is the analogue of Hoare's while rule for sequential programs (Hoare 1969).

5. Compositionality and real-time

In this chapter we adapt the compositional Hoare-style proof systems from the previous chapters to real-time. We describe in detail a compositional method to specify and verify timing constraints. By describing the details of a particular compositional proof method, we illustrate the general outline of such a description which should consist of the following points.

1. A description of the programming language, i.e., syntax and informal semantics.
2. A formal semantics of the programming language.
3. The definition of an assertion language in which properties of programs can be expressed. For this assertion language we also have to give syntax, informal meaning, and formal interpretation.
4. The definition of a correctness formula that relates programs and assertions. Using the semantics of the programming language and the interpretation of assertions, the validity of such a correctness formula can be defined formally.
5. A proof system in which, by rules and axioms, correctness formulae can be derived formally.
6. The proof of soundness and (relative) completeness of the proof system: show that every correctness formula that can be derived is also valid, and that every valid formula can be derived (assuming that valid assertions can be derived).

As an example, we consider in this section a compositional proof method for

distributed real-time systems based on Hooman (1991b). Concerning the points above, we describe:

1. A real-time programming language with nested parallelism, communication via synchronous message passing, and time-outs.
2. The semantic model which is used to give a denotational semantics for the programming language; the meaning of a program is given by a set of models where each model describes a possible computation of the program.
3. A first-order assertion language which includes primitives to specify the timing behaviour of programs.
4. A correctness formula of the form $C:\{p\}S\{q\}$.
5. A compositional proof system to derive these extended Hoare triples.
6. The proofs of soundness and (relative) completeness for the proof system given in this section are not given here. The reader is referred to Hooman (1991b) for all details about these proofs.

The syntax and informal semantics of our real-time programming language are given in § 5.1. A semantic model for this language is described in § 5.2. In § 5.3 we define the syntax and the interpretation of assertion language and correctness formulae. A compositional proof system for this formalism is presented in § 5.4.

5.1 Real-time programming language

Our real-time programming language is based on the Occam-like language from § 2 and is akin to real-time versions of CSP as defined in Koymans *et al* (1988) and Huizing *et al* (1987). We add a real-time statement **delay** d which suspends the execution for (at least) d time units. This statement is also used in the language Ada (Ada 1983) and corresponds to a *wait* d statement (Koymans *et al* 1988; Huizing *et al* 1987). Similar to a delay-statement in select construct of Ada, such a delay-statement is allowed in a guard of a guarded command to enable the programming of time-outs.

To investigate the basic real-time framework and to highlight the main points, no program variables are used – we consider only the (real-time) communication behaviour. In Hooman (1991b) we show that this framework can be extended to a language with program variables. Processes communicate and synchronize by message passing via unidirectional channels, each connecting two processes. Communication is synchronous.

5.1a Syntax and informal meaning: Let $TIME$ be some countable ordered time domain and ∞ a special symbol, $\infty \notin TIME$. The syntax of our programming language is given in table 2, with $n \in \mathbb{N}$, $c, c_1, \dots, c_n \in CHAN$, $d \in TIME$, and $d_0 \in TIME \cup \{\infty\}$, $d_0 > 0$.

Table 2. Syntax programming language.

<i>Statement</i>	$S ::= \text{skip} \mid \text{delay } d \mid c! \mid c? \mid$ $S_1; S_2 \mid G \mid \star G \mid S_1 \parallel S_2$
<i>Guarded command</i>	$G ::= [\bigwedge_{i=1}^n c_i? \rightarrow S_i \mid \text{delay } d_0 \rightarrow S]$

Since we have slightly modified the syntax of our programming language, we briefly mention the informal meaning of statements.

Atomic statements

- **skip** terminates immediately.
- **delay** d suspends execution for d time units.
- $c!$ is used to send a signal along channel c . (In this chapter we do not consider the value transmitted.) Since we are assuming synchronous communication, a statement $c!$ is suspended until the receiving process executes a statement $c?$.
- $c?$ is used to receive a signal along channel c . An input statement $c?$ is suspended until the sending process executes an output statement $c!$.

Compound statements

- $S_1; S_2$ indicates sequential composition of statements S_1 and S_2 .
- Guarded command $[\bigvee_{i=1}^n c_i? \rightarrow S_i \square \text{delay } d \rightarrow S]$ is executed as follows: wait at most d time units for some input guard $c_i?$ to become enabled, that is, until communication can actually occur along one of the c_i because a communication partner becomes available. If at least one of the c_i -communications is possible (before d time units have elapsed), one of these communications (non-deterministically chosen) is performed and thereafter the corresponding S_i is executed. If $d \neq \infty$ and no guard is enabled within d time units after the start of the execution of the command, then S is executed.

Example 5.1. This construct makes it possible to model a *time-out*, i.e., to restrict the waiting period for certain communications. Consider the guarded command $[c?x \rightarrow S_1 \square \text{delay } 5 \rightarrow S_2]$; if there is no partner available for the input statement within 5 time units then the delay-alternative is taken and S_2 is executed. $\square$

- $\star G$ indicates repeated execution of guarded command G . Since we do not consider boolean guards in this chapter, execution of $\star G$ never terminates.
- $S_1 \parallel S_2$ indicates parallel execution of S_1 and S_2 .

We often use $[\bigvee_{i=1}^n c_i? \rightarrow S_i]$ as an abbreviation of $[\bigvee_{i=1}^n c_i? \rightarrow S_i \square \text{delay } \infty \rightarrow S]$, if $n > 0$. Let $DCHAN$ be the set of channels extended with directional channels;

$$DCHAN = CHAN \cup \{c! | c \in CHAN\} \cup \{c? | c \in CHAN\}.$$

DEFINITION 5.1 (Channels occurring in statement)

The set of (directional) channels occurring in a statement S , notation $dch(S)$, is defined as the smallest subset of $DCHAN$ such that if c is an output channel of S then $\{c, c!\} \subseteq dch(S)$, and if c is an input channel of S then $\{c, c?\} \subseteq dch(S)$.

5.1b Syntactic restrictions: A number of syntactic constraints are imposed upon statements to guarantee that a channel connects exactly two processes. With the definition of $dch(S)$ above we can express these restrictions formally as follows:

- For $S_1; S_2$ we require that, for all $c \in CHAN$, $c! \in dch(S_1)$ implies $c? \notin dch(S_2)$, and $c? \in dch(S_1)$ implies $c! \notin dch(S_2)$.
- For a guarded command $G \equiv [\bigvee_{i=1}^n c_i? \rightarrow S_i \square \text{delay } d \rightarrow S_0]$ we require
 - for all $i \in \{1, \dots, n\}$ that $c_i! \notin dch(G)$, and
 - for all $i, j \in \{0, 1, \dots, n\}$, $i \neq j$, and $c \in CHAN$ that $c! \in dch(S_i)$ implies $c? \notin dch(S_j)$, and $c? \in dch(S_i)$ implies $c! \notin dch(S_j)$.
- For $S_1 \parallel S_2$ we require $dch(S_1) \cap dch(S_2) \subseteq CHAN$.

5.1c Basic timing assumptions: The precise meaning of this programming language is defined by a denotational semantics which describes the real-time behaviour of programs. Such real-time semantics requires information about implementation details from which one usually abstracts in non-real-time models, such as the execution time of assignments and the time required to evaluate boolean tests. Thus we have to make assumptions about the execution time of atomic statements and about the extra time needed to execute compound constructs, i.e., how the execution time of compound constructs can be obtained from the timing of the components. In our proof systems the correctness of a program with respect to a specification, which may include timing constraints, is verified relative to these assumptions.

In general we will have bounds on the execution time. Here we assume, for simplicity, that there is no overhead for compound statements and that a **delay** d statement takes exactly d time units. Furthermore we assume given a constant $K_c > 0$ such that each communication, i.e., without the waiting period, takes K_c time units.

The most important assumption involves parallel composition. To determine the execution time of parallel programs we need information about the progress of actions, representing the allocation of processes on processors. In general, we have to make an assumption about the execution model of parallel processes. In this paper we consider the *maximal parallelism* model to represent the situation that every process has its own processor. Hence, a process never waits with the execution of a local, non-communication, statement. An input or output statement can cause the process to wait, but only when no communication partner is available; as soon as both partners are available the communication must take place. Thus the maximal parallelism model implies a *minimal waiting period*.

In Hooman (1991a) we have generalized this maximal parallelism assumption to multiprogramming where several processes can be executed on a single processor and scheduling is based on priorities which can be assigned to statements in the program.

5.2 Semantic model

Our formal model of real-time communication behaviour consists of a mapping from points of time to sets of channel names, indicating the channels along which messages are being transmitted at any given time. In addition to the names of the channels along which a communication takes place, the model includes information about those processes waiting to send or waiting to receive messages on their incident channels at any given time. Using this information, the formalism enforces minimal waiting in our maximal parallelism model by requiring that no pair of processes is ever simultaneously waiting to send and waiting to receive, respectively, on a shared channel.

We express the timing behaviour of a program from the viewpoint of an external observer with his own clock (as done in Reed & Roscoe 1986, Koymans *et al* 1988). 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 a real-time computation of a program by means of a function which assigns to a point of time a set of records, representing the events that are taking place at that point.

In this paper 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 dense time domain a

communication can be represented by an interval of time points during which we record this communication, and we can easily model communications that overlap in time or that are arbitrarily close to each other in time. Having dense time is also suitable for the description of reactive systems which interact with an environment that has a time-continuous nature (see, e.g. Koymans 1990). Furthermore, we argue that in a compositional framework it is inconvenient to use discrete time. Recall that compositionality allows us to design a process in isolation according to its specification. With a discrete notion of time a smallest time unit has to be chosen in this specification, and then when two independently developed processes with different time units are combined, a new basic time unit must be defined and the specifications of the processes have to be modified accordingly. Finally, a dense time domain allows the refinement of a single event into a sequence of sub-events, such as the implementation of a single synchronous communication by a sequence of asynchronous communications according to some protocol. An extensive discussion about the nature of time can be found in Joseph & Goswami (1989). In this paper we use the non-negative rationals as our (dense) time domain:

$$TIME = \{\tau \in \mathcal{Q} \mid \tau \geq 0\},$$

where $\mathcal{Q}$ is the set of rational numbers.

For notational convenience, a special value ∞ is used with the following properties: $\infty \notin TIME$, for all $\tau \in TIME$: $\tau < \infty$, and for all $\tau \in TIME \cup \{\infty\}$: $\tau + \infty = \infty + \tau = \infty$, $\infty - \tau = \infty$, $\tau \times \infty = \infty \times \tau = \infty$, $\max(\infty, \tau) = \max(\tau, \infty) = \infty$, $\min(\infty, \tau) = \min(\tau, \infty) = \tau$, and $\min \phi = \infty$. For a point $\tau_0 \in TIME$, a left-closed right-open interval $[0, \tau_0)$ is defined as $\{\tau \in TIME \mid 0 \leq \tau < \tau_0\}$.

DEFINITION 5.2 (Model)

Let $\tau_0 \in TIME \cup \{\infty\}$.

A *model* σ (of a real-time computation) is a mapping $\sigma: [0, \tau_0) \rightarrow \not\!\!/(DCHAN)$.

DEFINITION 5.3 (Length of a model)

For a model σ with domain $[0, \tau_0)$ the *length* of σ , denoted by $|\sigma|$, is defined as $|\sigma| = \tau_0$.

Thus, for all $\tau \in TIME$, with $\tau < |\sigma|$, we have $\sigma(\tau) \subseteq DCHAN$. Informally, a model σ represents the communication behaviour at each point in time during an execution of a program. If $|\sigma| = \infty$ then σ represents a non-terminating computation, and if $|\sigma| < \infty$ then it represents a computation that terminates at time $|\sigma|$. For a point of time τ , $\tau < |\sigma|$, and a channel name $c \in CHAN$, we have three possible elements c , $c!$ and $c?$ for $\sigma(\tau)$ with the following meaning:

- $c \in \sigma(\tau)$ if a communication takes place along channel c at time τ ;
- $c! \in \sigma(\tau)$ if a process is waiting to send along channel c at time τ ;
- $c? \in \sigma(\tau)$ if a process is waiting to receive along channel c at time τ .

DEFINITION 5.4 (Concatenation of models)

Define the *concatenation* of two models σ_1 and σ_2 , denoted by $\sigma_1 \sigma_2$, as

$$|\sigma_1 \sigma_2| = |\sigma_1| + |\sigma_2|$$

and

$$\sigma_1 \sigma_2(\tau) = \begin{cases} \sigma_1(\tau) & \text{for all } \tau < |\sigma_1| \\ \sigma_2(\tau - |\sigma_1|) & \text{for all } |\sigma_1| \leq \tau < |\sigma_1| + |\sigma_2| \end{cases}$$

Note that, for all models σ_1 and σ_2 , if $|\sigma_1| = \infty$ then $\sigma_1 \sigma_2 = \sigma_1$.

A compositional semantics for our programming language is defined in Hooman (1991b) using the computational model as described above. The meaning of a program S , denoted by $\mathcal{M}(S)$, is a set of models representing the possible computations of S starting at time 0. $\mathcal{M}(S)$ is defined by induction on the structure of S according to the grammar in table 2.

5.3 Specifications

We modify the Hoare-style framework of the previous chapters by extending the first-order assertion language with primitives to specify the timing behaviour of programs. As already explained, by means of Hoare triples we can only specify partial correctness. Hence, we add a third assertion, called *commitment*, to specify real-time properties of terminating *and* non-terminating computations. In contrast with the previous chapters, the aim is to specify, besides safety properties (which can be falsified in infinite time), also liveness properties. Therefore the commitment will not be an invariant which holds at any point during a computation (as in §4), but it should hold for complete, possibly infinite, computations.

5.3a Modification of Hoare triples to real-time: To extend a Hoare triple $\{p\}S\{q\}$ to real-time, a special variable *time* is introduced. Consider, for instance the formula $\{time = 3\} \text{ delay } 2 \{time = 5\}$. In the pre-condition the variable *time* specifies the starting time of the program, whereas in the post-condition *time* denotes the termination time. Furthermore, to specify the timed communication behaviour of programs, the assertion language includes a primitive *comm via c at exp* to express that a communication along channel *c* takes place at time *exp*. As in the semantics, primitives are required to express that a process is waiting to communicate. Here we use *wait to c! at exp* to denote that a process is waiting to send a message along channel *c* at time *exp*, and *wait to c? at exp* to denote that a process is waiting to receive along channel *c* at time *exp*. As usual in Hoare-style formalisms, logical variables are used to relate pre- and post-condition. In this chapter we have logical variables ranging over $TIME \cup \{\infty\}$, and quantification over these variables. For instance, with logical variable *t*, the specification $\{time = t\}S\{t + 4 < time < t + 7\}$ expresses that if *S* terminates then it takes between 4 and 7 time units.

Recall that a formula $\{p\}S\{q\}$ can only express the behaviour of terminating computations, and hence such a specification is trivially satisfied by non-terminating programs. Therefore we extend a Hoare triple $\{p\}S\{q\}$ with a third assertion, called a *commitment*, which expresses the real-time communication behaviour of all executions of *S*, including the non-terminating ones. This leads to a correctness formula of the form $C: \{p\}S\{q\}$. In general, commitment *C* reflects the real-time communication interface between parallel components, whereas the pre- and post-condition facilitate the reasoning for sequential composition and iteration.

Finally, we argue that termination should be expressible in commitments. Consider the statements $S_1 \equiv c?$ and $S_2 \equiv [c? \rightarrow \text{skip} \parallel c? \rightarrow \star[\text{delay } 1 \rightarrow \text{skip}]]$. Then the programs

$S_1; d!$ and $S_2; d!$, which satisfy the same Hoare triples, can be distinguished in our extended framework by the commitment $\forall t_0: (comm \text{ via } c \text{ at } t_0 \rightarrow \exists t_1 \geq t_0: [wait \text{ to } d! \text{ at } t_1 \vee comm \text{ via } d \text{ at } t_1])$. Since we aim at a compositional proof system, the distinction between $S_1; d!$ and $S_2; d!$ implies that S_1 and S_2 must also be distinguishable. Hence we have to express termination in the commitment. This can be done conveniently, without introducing new primitives, by allowing the special variable *time* to occur in commitments. Observe that the commitment can be seen as an extension of the post-condition to non-terminating computations. Hence, by interpreting *time* similar to post-conditions, *time* in commitments expresses the termination time of terminating computations. For non-terminating computations we use the special variable ∞ and such computations satisfy the commitment $time = \infty$. In the example above, S_1 and S_2 can be distinguished by using the commitment $\forall t_0: (comm \text{ via } c \text{ at } t_0 \rightarrow time < \infty)$.

5.3b Assertion language: Let *TVAR* be a set of logical variables ranging over $TIME \cup \{\infty\}$. Thy syntax of the assertion language is given in table 3, where $t \in TVAR$, $c \in CHAN$, and $\tau \in TIME \cup \{\infty\}$. Let *dch*(*p*) denote the set of all *c*, *c!*, or *c?* occurring in assertion *p*.

Table 3. Syntax of the assertion language.

<i>Expression</i>	$exp ::= \tau \mid t \mid time \mid exp_1 + exp_2 \mid exp_1 \times exp_2$
<i>Assertion</i>	$p ::= comm \text{ via } c \text{ at } exp \mid wait \text{ to } c! \text{ at } exp \mid wait \text{ to } c? \text{ at } exp$ $exp_1 = exp_2 \mid exp_1 < exp_2 \mid exp \in \mathcal{N} \mid$ $\neg p \mid p_1 \vee p_2 \mid \exists t: p$

To interpret logical variables we use a logical variable environment $\gamma: TVAR \rightarrow TIME \cup \{\infty\}$, i.e., a mapping which assigns a value from $TIME \cup \{\infty\}$ to each logical variable. The value of a variable *t* in an environment γ is denoted by $\gamma(t)$. The *variant* of an environment γ with respect to a logical variable *t* and a value $\tau \in TIME \cup \{\infty\}$, denoted by $(\gamma: t \mapsto \tau)$, is defined as follows. For any logical variable t_1 ,

$$(\gamma: t \mapsto \tau)(t_1) = \begin{cases} \gamma(t_1) & \text{if } t \neq t_1 \\ \tau & \text{if } t \equiv t_1 \end{cases}$$

Then we formally define when an assertion *p* holds in an environment γ and a model σ as defined in § 5.2. The special variable *time* is interpreted as the length of σ (i.e., $|\sigma|$). If expression *exp* yields a value $\tau < |\sigma|$ then the interpretation of the primitives *comm via c at exp*, *wait to c! at exp*, and *wait to c? at exp* is straightforward using $\sigma(\tau)$. But if *exp* yields a value greater than $|\sigma|$ then we must be more careful with the meaning of these primitives. (Consider, for instance, *comm via c at (time + 3)*. If such an assertion would hold in a model, which is intuitively strange, then this would lead to problems in the proof system. For instance, for a formula $C: \{comm \text{ via } c \text{ at } (time + 3)\} S\{q\}$ the information from the pre-condition should not be used in *C* and *q*. In general, a pre-condition should express the behaviour before the start of a program and it should not restrict the behaviour of the program at points of time after the starting time. Thus we aim at an interpretation in which *comm via c at (time + 3)* never holds in a model. Note that $C: \{\neg comm \text{ via } c \text{ at } (time + 3)\} S\{q\}$ leads to the same problems, and hence also $\neg comm \text{ via } c \text{ at } (time + 3)$ should not hold in any model.

A possible solution is to be careful with negations and to apply it only to primitive assertions. Here we choose an alternative approach; to achieve a compositional definition of negation we use a three-valued interpretation. This means that the value

of an assertion p in an environment γ and a model σ , denoted $\llbracket p \rrbracket \gamma \sigma$, is *true*, *false*, or $\perp$. To define negation and disjunction of assertions, logical operators NOT_3 and OR_3 for these three values are defined by the truth tables in table 4. These operators, which were introduced in Kleene (1952), are the strongest monotonic extensions of the classical (two-valued) operators.

Table 4. Three-valued negation and disjunction.

p	$NOT_3 p$		OR_3	<i>true</i>	<i>false</i>	$\perp$
<i>true</i>	<i>false</i>		<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>		<i>false</i>	<i>true</i>	<i>false</i>	$\perp$
$\perp$	$\perp$		$\perp$	<i>true</i>	$\perp$	$\perp$

First we define the value of expression exp in a model σ and an environment γ , denoted $v(exp)(\gamma, \sigma)$, yielding a value from $TIME \cup \{\infty\}$.

- $v(\tau)(\gamma, \sigma) = \tau$,
- $v(t)(\gamma, \sigma) = \gamma(t)$,
- $v(time)(\gamma, \sigma) = |\sigma|$,
- $v(exp_1 + exp_2)(\gamma, \sigma) = v(exp_1)(\gamma, \sigma) + v(exp_2)(\gamma, \sigma)$,
- $v(exp_1 \times exp_2)(\gamma, \sigma) = v(exp_1)(\gamma, \sigma) \times v(exp_2)(\gamma, \sigma)$.

Next we define inductively $\llbracket p \rrbracket \gamma \sigma$ as an element of $\{true, false, \perp\}$.

- $\llbracket comm \text{ via } c \text{ at } exp \rrbracket \gamma \sigma = \begin{cases} true, & \text{if } v(exp)(\gamma, \sigma) < |\sigma| \text{ and } c \in \sigma(exp)(\gamma, \sigma) \\ false, & \text{if } v(exp)(\gamma, \sigma) < |\sigma| \text{ and } c \notin \sigma(v(exp)(\gamma, \sigma)), \\ \perp, & \text{if } v(exp)(\gamma, \sigma) \geq |\sigma| \end{cases}$
- $\llbracket wait \text{ to } c! \text{ at } exp \rrbracket \gamma \sigma = \begin{cases} true, & \text{if } v(exp)(\gamma, \sigma) < |\sigma| \text{ and } c! \in \sigma(v(exp)(\gamma, \sigma)) \\ false, & \text{if } v(exp)(\gamma, \sigma) < |\sigma| \text{ and } c! \notin \sigma(v(exp)(\gamma, \sigma)), \\ \perp, & \text{if } v(exp)(\gamma, \sigma) \geq |\sigma| \end{cases}$
- $\llbracket wait \text{ to } c? \text{ at } exp \rrbracket \gamma \sigma = \begin{cases} true, & \text{if } v(exp)(\gamma, \sigma) < |\sigma| \text{ and } c? \in \sigma(v(exp)(\gamma, \sigma)) \\ false, & \text{if } v(exp)(\gamma, \sigma) < |\sigma| \text{ and } c? \notin \sigma(v(exp)(\gamma, \sigma)), \\ \perp, & \text{if } v(exp)(\gamma, \sigma) \geq |\sigma| \end{cases}$
- $\llbracket exp_1 = exp_2 \rrbracket \gamma \sigma = \begin{cases} true, & \text{if } v(exp_1)(\gamma, \sigma) = v(exp_2)(\gamma, \sigma) \\ false, & \text{if } v(exp_1)(\gamma, \sigma) \neq v(exp_2)(\gamma, \sigma), \end{cases}$
- $\llbracket exp_1 < exp_2 \rrbracket \gamma \sigma = \begin{cases} true, & \text{if } v(exp_1)(\gamma, \sigma) < v(exp_2)(\gamma, \sigma) \\ false, & \text{if } v(exp_1)(\gamma, \sigma) \geq v(exp_2)(\gamma, \sigma), \end{cases}$
- $\llbracket exp \in \mathbb{N} \rrbracket \gamma \sigma = \begin{cases} true, & \text{if } v(exp)(\gamma, \sigma) \in \mathbb{N} \\ false, & \text{if } v(exp)(\gamma, \sigma) \notin \mathbb{N} \end{cases}$
- $\llbracket \neg p \rrbracket \gamma \sigma = NOT_3 \llbracket p \rrbracket \gamma \sigma$,
- $\llbracket p_1 \vee p_2 \rrbracket \gamma \sigma = \llbracket p_1 \rrbracket \gamma \sigma OR_3 \llbracket p_2 \rrbracket \gamma \sigma$,
- $\llbracket \exists t: p \rrbracket \gamma \sigma = \begin{cases} true, & \text{if there exists a } \tau \in TIME \cup \{\infty\} \text{ with } \llbracket p \rrbracket (\gamma: t \mapsto \tau) \sigma = true \\ false, & \text{if for all } \tau \in TIME \cup \{\infty\}, \llbracket p \rrbracket (\gamma: t \mapsto \tau) \sigma = false \\ \perp, & \text{otherwise.} \end{cases}$

The conventional abbreviations are used, such as $p_1 \wedge p_2 \equiv \neg(\neg p_1 \vee \neg p_2)$, $p_1 \rightarrow p_2 \equiv \neg p_1 \vee p_2$, and $\forall t: p \equiv \neg \exists t: \neg p$. Subsequently we say that p holds in γ and σ if $\llbracket p \rrbracket \gamma \sigma = \text{true}$. Furthermore, we frequently use $\llbracket p \rrbracket \gamma \sigma$ as an abbreviation of $\llbracket p \rrbracket \gamma \sigma = \text{true}$.

Returning to our example, observe that the interpretation is such that, for any γ and σ , $\llbracket \text{comm via } c \text{ at } (time + 3) \rrbracket \gamma \sigma = \perp$ and $\llbracket \neg \text{comm via } c \text{ at } (time + 3) \rrbracket \gamma \sigma = \perp$. Thus neither $\text{comm via } c \text{ at } (time + 3)$ nor $\neg \text{comm via } c \text{ at } (time + 3)$ holds in γ and σ . In general this interpretation facilitates sequential reasoning, since $\llbracket p \rrbracket \gamma \sigma$ implies that p does not express any constraint on points of time after $|\sigma|$. This is expressed formally in lemma 5.5 below: if assertion p holds in σ_1 then p , with $time$ replaced by $|\sigma_1|$, holds in any arbitrary extension of σ_1 , i.e., in $\sigma_1 \sigma_2$ for any σ_2 . (A proof for this lemma can be found in Hooman 1991b.)

Lemma 5.5. *For all γ and σ_1 : $\llbracket p \rrbracket \gamma \sigma_1$ iff (for all σ_2 , $\llbracket p[|\sigma_1|/time] \rrbracket \gamma \sigma_1 \sigma_2$).*

We use the conventional relations between expressions, such as

- $exp_1 \leq exp_2 \equiv (exp_1 < exp_2) \vee (exp_1 = exp_2)$,
- $exp_1 \geq exp_2 \equiv (exp_2 < exp_1) \vee (exp_1 = exp_2)$,
- $exp_1 \leq exp_2 \leq exp_3 \equiv (exp_1 \leq exp_2) \wedge (exp_2 \leq exp_3)$, etc.

Relativized quantifiers are defined as usual, for instance,

- $\forall t, t_0 \leq t < time: p \equiv \forall t: t_0 \leq t < time \rightarrow p$.
- $\exists t, t_0 \leq t < time: p \equiv \exists t: t_0 \leq t < time \wedge p$.

Furthermore, the following abbreviations are frequently used:

- $\text{true} \equiv 0 = 0$,
- $\text{false} \equiv \neg \text{true}$,
- $\text{wait to } c! \text{ during } [t_0, t_1) \equiv \forall t_2, t_0 \leq t_2 < t_1: \text{wait to } c! \text{ at } t_2$,
- $\text{comm via } c \text{ during } [t_0, t_1) \equiv \forall t_2, t_0 \leq t_2 < t_1: \text{comm via } c \text{ at } t_2$,
- $\text{no comm via } c \text{ during } [t_0, t_1) \equiv \forall t_2, t_0 \leq t_2 < t_1: \neg \text{comm via } c \text{ at } t_2$,
- $\text{wait to } c! \text{ at } t_0 \text{ until comm at } t_1 \equiv$
 $\text{wait to } c! \text{ during } [t_0, t_1) \wedge \text{comm via } c \text{ during } [t_1, t_1 + K_c)$,
- $\text{wait to } c! \text{ at } t_0 \text{ until comm} \equiv \exists t_1 \geq t_0: \text{wait to } c! \text{ at } t_0 \text{ until comm at } t_1$.

Let $cset$ be a finite subset of $DCHAN$. Then

- $\text{no } cset \text{ during } [t_0, t_1) \equiv \forall t, t_0 \leq t < t_1: \bigwedge_{c! \in cset} \neg \text{wait to } c! \text{ at } t \wedge$
 $\bigwedge_{c? \in cset} \neg \text{wait to } c? \text{ at } t \wedge \bigwedge_{c \in cset} \neg \text{comm via } c \text{ at } t$.

The abbreviations above are also used with $c?$ instead of $c!$, and with other intervals such as (t_0, t_1) and (t_0, ∞) instead of the interval $[t_0, t_1)$. It is easy to extend these definitions for general expressions instead of t_0 or t_1 .

Observe that logical variables range over $TIME \cup \{\infty\}$, and thus

$\text{wait to } c! \text{ at } t_0 \text{ until comm, i.e., } \exists t_1 \geq t_0: \text{wait to } c! \text{ at } t_0 \text{ until comm at } t_1$ is
equivalent to $[\text{wait to } c! \text{ at } t_0 \text{ until comm at } \infty] \vee [\exists t_1, t_0 \leq t_1 < \infty: \text{wait to } c! \text{ at } t_0 \text{ until comm at } t_1]$.

Since $\text{comm via } c \text{ during } [\infty, \infty + K_c) \leftrightarrow \text{true}$, this is equivalent to

$[\text{wait to } c! \text{ during } [t_0, \infty)) \vee$
 $[\exists t_1, t_0 \leq t_1 < \infty: \text{wait to } c! \text{ during } [t_0, t_1) \wedge \text{comm via } c \text{ during } [t_1, t_1 + K_c)]]$.

DEFINITION 5.6. (Validity assertions)

An assertion p is *valid*, denoted $\models p$, iff p holds in any environment γ and any model σ , i.e., $\llbracket p \rrbracket \gamma \sigma$ for all γ and σ .

Next we define when a correctness formula $C: \{p\} S \{q\}$ is valid. Informally, if p holds in an initial model $\hat{\sigma}$, and σ represents a computation of S , then C holds in the concatenation $\hat{\sigma}\sigma$ of these models, and if σ terminates then q holds in $\hat{\sigma}\sigma$. This leads to the following formal definition (recall that, e.g., $\llbracket p \rrbracket \gamma \hat{\sigma}$ is an abbreviation of $\llbracket p \rrbracket \gamma \hat{\sigma} = \text{true}$).

DEFINITION 5.7. (Validity of a correctness formula)

For a program S and assertions C , p and q , a correctness formula $C: \{p\} S \{q\}$ is *valid*, denoted by $\models C: \{p\} S \{q\}$, iff for any γ and any well-formed model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$:

if $\llbracket p \rrbracket \gamma \hat{\sigma}$ then for all $\sigma \in \mathcal{M}(S): \llbracket C \rrbracket \gamma(\hat{\sigma}\sigma)$, and if $|\hat{\sigma}| < \infty$ then $\llbracket q \rrbracket \gamma(\hat{\sigma}\sigma)$.

Example 5.2. We show that $\models \text{time} = t + d: \{\text{time} = t\} \text{delay } d \{\text{time} = t + d\}$. Consider an environment γ and a model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$. Assume $\llbracket \text{time} = t \rrbracket \gamma \hat{\sigma}$. Then $|\hat{\sigma}| = \gamma(t)$, i.e., the starting time is the value of t in environment γ . For $\sigma \in \mathcal{M}(\text{delay } d)$ we have $|\sigma| = d$. Then $\llbracket \text{time} = t + d \rrbracket \gamma \hat{\sigma}\sigma$, since $|\hat{\sigma}\sigma| = |\hat{\sigma}| + |\sigma| = \gamma(t) + d$. $\square$

Observe that the definition of validity of a correctness formula requires that the assertions hold for each environment γ , and hence free logical variables in a specification are implicitly universally quantified.

Example 5.3. As an example of a liveness specification, consider the formula

$$C \wedge \text{time} = \infty: \{\text{time} = 0\} \star [c? \rightarrow \text{skip}] \{\text{false}\},$$

with

$$C \equiv (\forall t_0 < \infty \exists t_1, > t_0: \text{comm via } c \text{ at } t_1) \vee (\exists t_0 < \infty: \text{wait to } c? \text{ during } [t_0, \infty)).$$

This commitment expresses that the program either communicates infinitely often, or it eventually waits forever. Observe that this is not a safety property since it cannot be falsified in finite time. After presenting the rule for the iteration construct we show that this valid formula is also derivable, as it should be in a complete proof system. $\square$

5.4 Proof system

In this section we give a compositional proof system for our correctness formulae. First we formulate rules and axioms that are generally applicable to any statement. Next we axiomatize the programming language by formulating rules and axioms for all atomic statements and compounds programming constructs. Let $\vdash C: \{p\} S \{q\}$ denote that the formula $C: \{p\} S \{q\}$ can be derived in this proof system.

General part

We start with an axiom expressing the well-formedness properties of a computation. Let cset be a finite subset of $DCHAN$.

Axiom 5.8. (Well-formedness)

$$Well\ Form_{cset} : \{true\} S \{ Well\ Form_{cset} \}$$

where

$$Well\ Form_{cset} \equiv \forall t < time : MW_{cset}(t) \wedge Excl_{cset}(t),$$

with

$$\begin{aligned} MW_{cset}(t) &\equiv \bigwedge_{\{c!, c?\} \subseteq cset} \neg (wait\ to\ c? \text{ at } t \wedge wait\ to\ c! \text{ at } t), \\ Excl_{cset}(t) &\equiv \left(\bigwedge_{\{c, c!\} \subseteq cset} \neg (wait\ to\ c! \text{ at } t \wedge comm\ via\ c \text{ at } t) \right) \wedge \\ &\quad \left(\bigwedge_{\{c, c?\} \subseteq cset} \neg (wait\ to\ c? \text{ at } t \wedge comm\ via\ c \text{ at } t) \right). \end{aligned}$$

We give a few lemmas concerning the use of $Well\ Form_{cset}$. These lemmas will be used in the chapter 6 where we illustrate our formalism by an example of a watchdog timer.

Lemma 5.9. For all t_0, t_1 ,

$$wait\ to\ c? \text{ during } (t_0, t_1) \wedge Well\ Form_{\{c, c!, c?\}} \rightarrow no\{c!, c\} \text{ during } (t_0, t_1).$$

Lemma 5.10. For all t_0 ,

$$\begin{aligned} wait\ to\ c! \text{ at } t_0 \text{ until } comm \wedge Well\ Form_{\{c, c!, c?\}} &\rightarrow \\ &\forall t_1, t_0 < t_1 < t_0 + K_c : \neg wait\ to\ c? \text{ at } t_1. \end{aligned}$$

The proof system contains a consequence rule which is an extension of the classical consequence rule for Hoare triples (see rule 3.3 in §3). Note that by definition 5.7 of a valid correctness formula, pre-conditions are interpreted in a model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$. Hence any pre-condition can be strengthened by adding $time < \infty$ to express that the starting time is finite.

Rule 5.11. (Consequence)

$$\frac{C_0 : \{p_0\} S \{q_0\}, p \wedge time < \infty \rightarrow p_0, C_0 \rightarrow C, q_0 \rightarrow q}{C : \{p\} S \{q\}}.$$

Observe that $\models false : \{time = \infty\} S \{false\}$, for any program S . To deduce this formula, we first derive $false : \{false\} S \{false\}$. This can be done by means of the initial invariance rule below. Then we can use the consequence rule, since $time = \infty \wedge time < \infty \rightarrow false$.

Next we give two axioms to deduce invariance properties. The first axiom expresses that the pre-condition, except for the variable $time$, remains valid during the execution of a program.

Axiom 5.12. (Initial invariance)

$$p : \{p\} S \{p\}$$

provided $time$ does not occur in p .

The soundness of this axiom is based on lemma 5.5 which guarantees that if $\llbracket p \rrbracket \gamma \hat{\sigma}$ and p does not contain *time*, then $\llbracket p \rrbracket \gamma \hat{\sigma}$, for any model σ .

The channel invariance axiom below expresses that during the execution of a program S no activity takes place on channels not occurring in S . Let $cset$ be a finite subset of $DCHAN$.

Axiom 5.13. (Channel invariance)

$$no\ cset\ during\ [t_0, time): \{time = t_0\} S \{no\ cset\ during\ [t_0, time)\}$$

provided $cset \cap dch(S) = \emptyset$.

Our proof system contains the following rules for conjunction, quantification, and substitution.

Rule 5.14. (Conjunction)

$$\frac{C_1: \{p_1\} S \{q_1\}, C_2: \{p_2\} S \{q_2\}}{C_1 \wedge C_2: \{p_1 \wedge p_2\} S \{q_1 \wedge q_2\}}.$$

Rule 5.15. (Quantification)

$$\frac{C: \{p\} S \{q\}}{C: \{\exists t: p\} S \{q\}}$$

provided t does not occur in C and q .

Rule 5.16. (Substitution)

$$\frac{C: \{p\} S \{q\}}{C[exp/t]: \{p[exp/t]\} S \{q[exp/t]\}}$$

provided *time* does not occur in expression *exp*.

The following rule can be used to transform a correctness formula with pre-condition $time = t_0$ into a formula with an arbitrary pre-condition and starting time. This is a derived rule, that is, the rule can be derived from the other rules and axioms in the proof system (as we prove below).

Derived rule 5.17. (Adaptation)

$$\frac{C: \{time = t_0\} S \{q\}}{p[exp/time] \wedge C[exp/t_0]: \{p[exp/time] \wedge time = exp\} S \{p[exp/time] \wedge q[exp/t_0]\}}$$

provided *time* does not occur in expression *exp*.

Proof. Assume $\vdash C: \{time = t_0\} S \{q\}$, and suppose *time* does not occur in expression *exp*. By the substitution rule, replacing t_0 by *exp*, we obtain

$$\vdash C[exp/t_0]: \{time = exp\} S \{q[exp/t_0]\}$$

Since *time* does not occur in *exp*, the initial invariance rule leads to

$$\vdash p[exp/time]:\{p[exp/time]\}S\{p[exp/time]\}$$

Then, by the consequence rule,

$$\vdash p[exp/time] \wedge C[exp/t_0]:\{p[exp/time] \wedge time = exp\} \\ S\{p[exp/time] \wedge q[exp/t_0]\} \quad \square$$

Program part

The rules and axioms for atomic statements will be given with pre-condition *time* = *t*₀; with the adaptation rule one can easily obtain any arbitrary pre-condition.

Axiom 5.18. (Skip)

$$time = t_0:\{time = t_0\}\mathbf{skip}\{time = t_0\}$$

Axiom 5.19. (Delay)

$$time = t_0 + d:\{time = t_0\}\mathbf{delay} d\{time = t_0 + d\}$$

Example 5.4. We can derive

$$(time = 5 \wedge comm \text{ via } c \text{ at } 1):\{time = 2 \wedge comm \text{ via } c \text{ at } 1\}\mathbf{delay} 3\{time = 5\}$$

as follows. By the delay axiom,

$$time = t_0 + 3:\{time = t_0\}\mathbf{delay} 3\{time = t_0 + 3\}.$$

Using the adaptation rule, with $p \equiv comm \text{ via } c \text{ at } 1$ and $exp = 2$, we obtain

$$(time = 2 + 3 \wedge comm \text{ via } c \text{ at } 1): \\ \{time = 2 \wedge comm \text{ via } c \text{ at } 1\}\mathbf{delay} 3\{time = 2 + 3 \wedge comm \text{ via } c \text{ at } 1\}.$$

Finally, the consequence rule leads to

$$(time = 5 \wedge comm \text{ via } c \text{ at } 1):\{time = 2 \wedge comm \text{ via } c \text{ at } 1\}\mathbf{delay} 3\{time = 5\}. \quad \square$$

To formulate a rule for a send statement *c!*, observe that the post-condition can characterize terminating computations consisting of a waiting period (during which no communication partner is available) followed by an interval during which the actual communication takes place. In addition, the commitment can characterize non-terminating computations in which the i/o-statement waits forever to communicate. Observe that $\exists t \geq t_0: wait \text{ to } c! \text{ at } t_0 \text{ until } comm \text{ at } t \wedge time = t + K_c$ implies that either $t = \infty$ and *wait to c!* during $[t_0, \infty) \wedge time = \infty$, or there exists a $t, t_0 \leq t < \infty$ such that *wait to c!* during $[t_0, t) \wedge comm \text{ via } c \text{ during } [t, t + K_c) \wedge time = t + K_c$. This leads to the following rule:

Rule 5.20. (Send)

$$\frac{\exists t \geq t_0: \text{wait to } c! \text{ at } t_0 \text{ until comm at } t \wedge \text{time} = t + K_c \rightarrow C}{C: \{\text{time} = t_0\} c! \{C \wedge \text{time} < \infty\}}$$

Similar to the send rule, we have the following rule for a receive statement.

Rule 5.21. (Receive)

$$\frac{\exists t \geq t_0: \text{wait to } c? \text{ at } t_0 \text{ until comm at } t \wedge \text{time} = t + K_c \rightarrow C}{C: \{\text{time} = t_0\} c? \{C \wedge \text{time} < \infty\}}.$$

The inference rule for sequential composition is an extension of the classical rule for Hoare triples. To explain the commitment of $S_1; S_2$, observe that a computation of $S_1; S_2$ is either a non-terminating computation of S_1 or a terminated computation of S_1 extended with a computation of S_2 . The commitment of $S_1; S_2$ expresses the non-terminating computations of S_1 by using the commitment of S_1 with $\text{time} = \infty$. Terminating computations of S_1 are characterized in the post-conditions of S_1 which is also the pre-condition of S_2 . Then these computations are extended by S_2 and described in the commitment of S_2 .

Rule 5.22. (Sequential composition)

$$\frac{C_1: \{p\} S_1 \{r\}, C_2: \{r\} S_2 \{q\}}{(C_1 \wedge \text{time} = \infty) \vee C_2: \{p\} S_1; S_2 \{q\}}.$$

Example 5.5. Consider the program $c?; d!$. Define

$$C_{\text{nonterm}}^1 \equiv \text{wait to } c? \text{ during } [0, \infty)_1 \text{ and } C_{\text{term}}^1 \equiv \text{wait to } c? \text{ at } 0 \text{ until comm}_2 \text{ at } t_1.$$

Then

$$(C_{\text{nonterm}}^1 \wedge \text{time} = \infty) \vee (\exists t_1 < \infty: C_{\text{term}}^1 \wedge \text{time} = t_1 + K_c): \{ \text{time} = 0 \} c? \{ \exists t_1 < \infty: C_{\text{term}}^1 \wedge \text{time} = t_1 + K_c \}.$$

For $d!$, define $C_2 \equiv \text{wait to } d! \text{ at } t_1 + K_c \text{ until comm}$. Then we can derive

$$(\exists t_1 < \infty: C_{\text{term}}^1 \wedge C_2): \{ \exists t_1 < \infty: C_{\text{term}}^1 \wedge \text{time} = t_1 + K_c \} d! \{ \text{true} \}.$$

Observe that the terminating behaviour of $c?$ is characterized by its post-condition, thus by the pre-condition of $d!$, and hence can be included in the commitment of $d!$. Now the sequential composition rule leads to

$$(C_{\text{nonterm}}^1 \wedge \text{time} = \infty) \vee (\exists t_1 < \infty: C_{\text{term}}^1 \wedge C_2): \{ \text{time} = 0 \} c?; d! \{ \text{true} \}. \quad \square$$

Given the rules for the basic statements, it is often easier to use the following derived rule:

Derived rule 5.23. (Sequential composition adaptation)

$$\frac{C_1: \{p\} S_1 \{r\}, C_2: \{ \text{time} = t \} S_2 \{q\}}{(C_1 \wedge \text{time} = \infty) \vee (\exists t: r[t/\text{time}] \wedge C_2): \{p\} S_1; S_2 \{ \exists t: r[t/\text{time}] \wedge q \}}$$

Proof. Assume

$$\vdash C_1 : \{p\} S_1 \{r\}, \quad (1)$$

$$\vdash C_2 : \{time = t\} S_2 \{q\}. \quad (2)$$

Since $r \rightarrow \exists t: r[t/time] \wedge time = t$, (1) leads by the consequence rule to

$$\vdash C_1 : \{p\} S_1 \{\exists t: r[t/time] \wedge time = t\}. \quad (3)$$

By (2) and the adaptation rule:

$$\vdash r[t/time] \wedge C_2 : \{r[t/time] \wedge time = t\} S_2 \{r[t/time] \wedge q\}.$$

The consequence rule leads to

$$\vdash (\exists t: r[t/time] \wedge C_2) : \{r[t/time] \wedge time = t\} S_2 \{\exists t: r[t/time] \wedge q\}.$$

Then, using the quantification rule, we obtain

$$\vdash (\exists t: r[t/time] \wedge C_2) : \{\exists t: r[t/time] \wedge time = t\} S_2 \{\exists t: r[t/time] \wedge q\}. \quad (4)$$

From (3) and (4), by the sequential composition rule,

$$\vdash (C_1 \wedge time = \infty) \vee (\exists t: r[t/time] \wedge C_2) : \{p\} S_1; S_2 \{\exists t: r[t/time] \wedge q\}. \quad \square$$

Example 5.6. Consider the program $c?; d!$. We prove

$$(\exists t_1 \geq t_0 : \text{wait to } c? \text{ at } t_0 \text{ until comm at } t_1 \wedge \text{wait to } d! \text{ at } t_1 + K_c \text{ until comm}) : \\ \{time = t_0\} c?; d! \{true\}$$

Let $C_1 \equiv \exists t_1 \geq t_0 : \text{wait to } c? \text{ at } t_0 \text{ until comm at } t_1 \wedge time = t_1 + K_c$, then by the receive rule we can derive

$$C_1 : \{time = t_0\} c? \{C_1 \wedge time < \infty\}.$$

Let $C_2 \equiv \text{wait to } d! \text{ at } t \text{ until comm}$, then from the send rule we obtain

$$C_2 : \{time = t\} d! \{true\}.$$

By the derived sequential composition rule we can now derive

$$(C_1 \wedge time = \infty) \vee (\exists t: (C_1 \wedge time < \infty)[t/time] \wedge C_2) : \\ \{time = t_0\} c?; d! \{\exists t: (C_1 \wedge time < \infty)[t/time] \wedge true\}.$$

Observe that the commitment $(C_1 \wedge time = \infty) \vee (\exists t: (C_1 \wedge time < \infty)[t/time] \wedge C_2)$ implies

$$[\text{wait to } c? \text{ at } t_0 \text{ until comm at } \infty \wedge time = \infty] \vee \\ [\exists t \exists t_1 \geq t_0 : \text{wait to } c? \text{ at } t_0 \text{ until comm at } t_1 \wedge t = t_1 + K_c \wedge t < \infty \wedge \\ \text{wait to } d! \text{ at } t \text{ until comm}],$$

and thus

$$[\text{wait to } c? \text{ at } t_0 \text{ until comm at } \infty \wedge time = \infty] \vee \\ [\exists t_1, t_0 \leq t_1 < \infty : \text{wait to } c? \text{ at } t_0 \text{ until comm at } t_1 \wedge \text{wait to } d! \text{ at } t_1 + K_c \text{ until comm}].$$

Since $\models \text{wait to } d! \text{ at } \infty \text{ until comm}$, the consequence rule leads to

$$(\exists t_1 \geq t_0: \text{wait to } c? \text{ at } t_0 \text{ until comm at } t_1 \wedge \text{wait to } d! \text{ at } t_1 + K_c \text{ until comm});$$

$$\{time = t_0\} c?; d! \{true\}$$

□

Consider a guarded command $G \equiv [\bigwedge_{i=1}^n c_i? \rightarrow S_i \square \text{delay } d \rightarrow S]$. Define

- $\text{wait in } G \text{ during } [t_0, t) \equiv \bigwedge_{i=1}^n \text{wait to } c_i? \text{ during } [t_0, t) \wedge \text{no}(dch(G) - \{c_1?, \dots, c_n?\}) \text{ during } [t_0, t)$
- $\text{comm } c_i \text{ in } G \text{ from } t \equiv \text{comm via } c_i \text{ during } [t, t + K_c) \wedge \text{no}(dch(G) - \{c_i\}) \text{ during } [t, t + K_c) \wedge time = t + K_c$

First we give a rule for the case that $d = \infty$, thus for $G \equiv [\bigwedge_{i=1}^n c_i? \rightarrow S_i]$. This statement either waits forever to perform one of the $c_i?$ communications because none of the partners is available, or it eventually communicates via one of the $c_i?$ and then executes the corresponding statement S_i .

Rule 5.24. (Guarded command without delay)

$$\begin{array}{l} \text{wait in } G \text{ during } [t_0, \infty) \wedge time = \infty \rightarrow C_{\text{nonterm}} \\ \exists t, t_0 \leq t < \infty: \text{wait in } G \text{ during } [t_0, t) \wedge \text{comm } c_i \text{ in } G \text{ from } t \rightarrow p_i, \\ \text{for all } i = 1, \dots, n \end{array}$$

$$\frac{C_i: \{p_i\} S_i \{q_i\}, \text{ for all } i = 1, \dots, n}{C_{\text{nonterm}} \vee \bigvee_{i=1}^n C_i: \{time = t_0\} [\bigwedge_{i=1}^n c_i? \rightarrow S_i] \{ \bigvee_{i=1}^n q_i \}}$$

Next consider $G \equiv [\bigwedge_{i=1}^n c_i? \rightarrow S_i \square \text{delay } d \rightarrow S]$ with $d \neq \infty$.

Rule 5.25. (Guarded command with delay)

$$\begin{array}{l} \exists t, t_0 \leq t < t_0 + d: \text{wait in } G \text{ during } [t_0, t] \wedge \\ \text{comm } c_i \text{ in } G \text{ from } t \rightarrow p_i, \text{ for all } i = 1, \dots, n \\ C_i: \{p_i\} S_i \{q_i\}, \text{ for all } i = 1, \dots, n \\ C: \{ \text{wait in } G \text{ during } [t_0, t_0 + d) \wedge time = t_0 + d \} S \{q\} \\ \hline \bigvee_{i=1}^n C_i \vee C: \{time = t_0\} [\bigwedge_{i=1}^n c_i? \rightarrow S_i \square \text{delay } d \rightarrow S] \{ \bigvee_{i=1}^n q_i \vee q \} \end{array}$$

provided $d \neq \infty$.

The rule for the iteration construct does not contain any explicit well-foundedness argument, although we deal with liveness properties. The main principle is that liveness properties can be derived from real-time safety properties, and these properties can be proved by means of an invariant (see example 5.7),

Rule 5.26. (Iteration)

$$\begin{array}{l} C: \{C\} G \{C\}. \\ \hline (\forall t_1 < \infty \exists t_2 > t_1: C[t_2/time]) \rightarrow C_{\text{nonterm}} \\ C_{\text{nonterm}} \wedge time = \infty: \{C\} \star G \{false\} \end{array}$$

where t_1 and t_2 are fresh logical variables.

Example 5.7. Consider the formula from example 5.3, expressing a liveness property for program $\star[c? \rightarrow \text{skip}]$. Let $C_1 \equiv \forall t_3 < \infty \exists t_4 > t_3: \text{comm via } c \text{ at } t_4$ and

$$C_2 \equiv \exists t_3 < \infty: \text{wait to } c? \text{ during } [t_3, \infty).$$

To prove $(C_1 \vee C_2) \wedge \text{time} = \infty: \{ \text{time} = 0 \} \star[c? \rightarrow \text{skip}] \{ \text{false} \}$, we apply the iteration rule with $C_{\text{nonterm}} \equiv C_1 \vee C_2$ and $C \equiv (\forall t_3 < \text{time} \exists t_4 > t_3: \text{comm via } c \text{ at } t_4) \vee (\exists t_3 < \text{time}: \text{wait to } c? \text{ during } [t_3, \infty))$.

Observe that C expresses that $C_1 \vee C_2$ holds up to the termination time. We show that the two conditions of the iteration rule are fulfilled:

1. We prove $C: \{C\} [c? \rightarrow \text{skip}] \{C\}$ as follows.

By the rule for guarded command without delay we obtain

$$\hat{C}: \{ \text{time} = t_0 \} [c? \rightarrow \text{skip}] \{ \hat{C} \},$$

with

$$\hat{C} \equiv (\forall t_5, t_0 \leq t_5 < \text{time} \exists t_6 > t_5: \text{comm via } c \text{ at } t_6) \vee (\exists t_5, t_0 \leq t_5 < \text{time}: \text{wait to } c? \text{ during } [t_5, \infty)).$$

From the adaptation rule, with $p \equiv C$ and $\text{exp} \equiv t_0$,

$$C[t_0/\text{time}] \wedge \hat{C}: \{C[t_0/\text{time}] \wedge \text{time} = t_0\} [c? \rightarrow \text{skip}] \{C[t_0/\text{time}] \wedge \hat{C}\}.$$

Since $C[t_0/\text{time}] \wedge \hat{C} \rightarrow C$, the consequence rule leads to

$$C: \{C[t_0/\text{time}] \wedge \text{time} = t_0\} [c? \rightarrow \text{skip}] \{C\}.$$

By the quantification rule,

$$C: \{ \exists t_0: C[t_0/\text{time}] \wedge \text{time} = t_0 \} [c? \rightarrow \text{skip}] \{C\}.$$

Since $C \rightarrow \exists t_0: C[t_0/\text{time}] \wedge \text{time} = t_0$, the consequence rule leads to

$$C: \{C\} [c? \rightarrow \text{skip}] \{C\}.$$

2. We have $(\forall t_1 < \infty \exists t_2 > t_1: C[t_2/\text{time}]) \rightarrow C_{\text{nonterm}}$, since

$$\begin{aligned} (\forall t_1 < \infty \exists t_2 > t_1: C[t_2/\text{time}]) &\equiv (\forall t_1 < \infty \exists t_2 > t_1: \\ &[\forall t_3 < t_2 \exists t_4 > t_3: \text{comm via } c \text{ at } t_4] \vee [\exists t_3 < t_2: \text{wait to } c? \text{ during } [t_3, \infty)]) \rightarrow \\ &([\forall t_3 < \infty \exists t_4 > t_3: \text{comm via } c \text{ at } t_4] \vee [\exists t_3 < \infty: \text{wait to } c? \text{ during } [t_3, \infty)]) \equiv \\ &(C_1 \vee C_2) \equiv C_{\text{nonterm}}. \end{aligned}$$

Now by the iteration rule we obtain $(C_1 \vee C_2) \wedge \text{time} = \infty: \{C\} \star[c? \rightarrow \text{skip}] \{ \text{false} \}$. Since logical time-variables such as t_3 and t_4 range over nonnegative values, $\text{time} = 0$ implies $\forall t_3 < \text{time} \exists t_4 > t_3: \text{comm via } c \text{ at } t_4$, and hence $\text{time} = 0 \rightarrow C$. Thus, by the consequence rule, $(C_1 \vee C_2) \wedge \text{time} = \infty: \{ \text{time} = 0 \} \star[c? \rightarrow \text{skip}] \{ \text{false} \}$. $\square$

Consider the parallel composition of statements S_1 and S_2 . For the pre-conditions we simply take the conjunction. For the post-condition q of $S_1 \parallel S_2$ we would also prefer to take the conjunction of the post-conditions q_1 and q_2 of, respectively S_1 and S_2 , but there a small problem has to be solved. Observe that, for $i = 1, 2$, the special variable time in post-condition q_i of S_i denotes the termination time of S_i . Since, in

general, the termination times of S_1 and S_2 will be different (and then $q_1 \wedge q_2$ could imply *false*, see example 5.8), we substitute a logical variable t_i for *time* in q_i . Then the termination time of $S_1 \parallel S_2$, expressed by *time* in its post-condition, is the maximum of t_1 and t_2 . Furthermore we add two predicates to express that process S_i does not perform any action between t_i and *time*. A similar construction is used for the commitments. This leads to the following rule:

Rule 5.27. (Parallel composition)

$$\frac{C_i:\{p_i\}S_i\{q_i\}, i=1,2 \quad \begin{array}{l} \exists t_1, t_2: \text{time} = \max(t_1, t_2) \wedge \bigwedge_{i=1}^2 C_i[t_i/\text{time}] \wedge \text{no } dch(S_i) \text{ during } [t_i, \text{time}] \rightarrow C \\ \exists t_1, t_2: \text{time} = \max(t_1, t_2) \wedge \bigwedge_{i=1}^2 q_i[t_i/\text{time}] \wedge \text{no } dch(S_i) \text{ during } [t_i, \text{time}] \rightarrow q \end{array}}{C:\{p_1 \wedge p_2\}S_1 \parallel S_2\{q\}}$$

provided t_1 and t_2 are fresh logical variables, and $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$.

Example 5.8. To illustrate the problem with the termination times at parallel composition, consider the following two (valid) formulae:

$$\text{time} = 5: \{ \text{time} = 0 \} \text{delay } 5 \{ \text{time} = 5 \},$$

and

$$\text{time} = 7: \{ \text{time} = 0 \} \text{delay } 7 \{ \text{time} = 7 \}.$$

Then for **delay 5** $\parallel$ **delay 7** we cannot take the conjunction of commitments and post-conditions, but by the rule above we obtain the commitment and post-condition $\text{time} = 7$ because $(\exists t_1, t_2: \text{time} = \max(t_1, t_2) \wedge t_1 = 5 \wedge t_2 = 7) \rightarrow (\text{time} = 7)$. $\square$

If *time* does not occur in commitments and post-conditions of the components S_1 and S_2 then we can derive from rule 5.27 the following simple rule:

Derived rule 5.28. (Simple parallel composition)

$$\frac{C_1:\{p_1\}S_1\{q_1\}, C_2:\{p_2\}S_2\{q_2\}}{C_1 \wedge C_2:\{p_1 \wedge p_2\}S_1 \parallel S_2\{q_1 \wedge q_2\}}$$

provided $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$, and *time* does not occur in C_1, C_2, q_1 , and q_2 .

6. Example – watchdog timer

The formalism from §5 is illustrated by an example of a watchdog timer. Consider the network pictured in figure 1. Process W is a “watchdog” process: its job is to ensure that processes $P_1, \dots, P_n$ are functioning properly. We abstract from the task that has to be performed by P_i , but we assume that P_i is functioning correctly iff it is ready to send (or sending) a reset signal on channel re_i to W at least once every v_i time units, for some constant v_i . So long as all processes P_i are ready to send a reset signal in time, watchdog timer W communicates on each re_i at least once every v_i time units and then it does not communicate on channel *al*. As soon as W has to

Figure 1. Watchdog timer network.

wait for a reset signal on a particular re_k during v_k time units, then it is ready to send (or sending) an alarm message on channel al within, say, K time units.

In this section we first give a formal specification for process W . Then, given specifications for the P_i , we prove that $P_1 \parallel \dots \parallel P_n \parallel W$ is ready to send (or sending) on channel al iff one of the P_i is not functioning correctly. This is verified using our proof system without knowing the implementations of $P_1, \dots, P_n$ and W . To demonstrate program design from a specification, W is implemented as a parallel composition $W_1 \parallel \dots \parallel W_n \parallel A$, and we derive the specification of W using specifications for W_i and A . Next W_i and A are, independently, implemented, and we prove that these programs satisfy the corresponding specifications.

6.1 Specification of the watchdog timer

We give a formal specification for the watchdog timer W and derive properties from it, using certain specifications for the processes P_i . In the specification of W we express that if there is a waiting period of v_k time units to receive input via re_k then, for some constant K , W starts waiting to send on channel al within K until the actual communication takes place. Furthermore, W tries to communicate via channel al at a certain point of time only if, for some k , there was a previous period of at least v_k time units during which W is waiting to receive input via re_k . Let

$$\begin{aligned} C_0^W &\equiv \forall t_0 < \infty : \text{wait to } re_k? \text{ during } (t_0, t_0 + v_k) \rightarrow \\ &\quad (\exists t \leq t_0 + v_k + K : \text{wait to } al! \text{ at } t \text{ until comm}), \\ C_1^W &\equiv \forall t_1 < \infty : \text{wait to } al! \text{ at } t_1 \text{ until comm} \rightarrow \\ &\quad (\exists k \exists t_2 \leq t_1 : \text{wait to } re_k? \text{ during } (t_2, t_2 + v_k)). \end{aligned}$$

Then we specify W by $C_0^W \wedge C_1^W : \{time = 0\} W \{true\}$.

We prove that W tries to send a message via al iff there is an error in one of the processes P_i . Therefore we assume given a specification for P_i in which we use a predicate $error_i$ representing some erroneous behaviour of P_i . Thus assume that, for all i , we have $C^{P_i} : \{time = 0\} P_i \{true\}$, where

$$C^{P_i} \equiv error_i \leftrightarrow (\exists t_0 < \infty : \text{no } \{re_i!, re_i\} \text{ during } (t_0, t_0 + v_i)).$$

This asserts that there is an error in P_i iff there exists a period of v_i time units during which P_i is not communicating via re_i and not waiting to communicate via re_i . Given our specifications for $P_1, \dots, P_n$ and W , we try to prove that $P_1 \parallel \dots \parallel P_n \parallel W$ satisfies

the commitment $(\exists k: error_k) \leftrightarrow (\exists t < \infty: wait\ to\ all\ at\ t\ until\ comm)$. Applying the simple parallel composition rule n times we obtain

$$C_0^W \wedge C_1^W \wedge \bigwedge_{i=1}^n C^{P_i}: \{time = 0\} P_1 \parallel \dots \parallel P_n \parallel W \{true\}.$$

By the well-formedness axiom and the consequence rule we can derive

$$Well\ Form_{\{re_k, re_k!, re_k? | k = 1, \dots, n\}}: \{true\} P_1 \parallel \dots \parallel P_n \parallel W \{true\}.$$

Using the conjunction rule we obtain the following commitment:

$$C_0^W \wedge C_1^W \wedge \bigwedge_{i=1}^n C^{P_i} \wedge Well\ Form_{\{re_k, re_k!, re_k? | k = 1, \dots, n\}}.$$

First we prove that this commitment implies

$$\begin{aligned} & (\exists t < \infty: wait\ to\ all\ at\ t\ until\ comm) \rightarrow (\exists k: error_k). \\ & \quad \exists t < \infty: wait\ to\ all\ at\ t\ until\ comm \\ \Rightarrow \{C_1^W\} & \quad \exists t < \infty \exists k \exists t_2 \leq t: wait\ to\ re_k? \text{ during } (t_2, t_2 + v_k) \\ \Rightarrow \{\text{calculus}\} & \quad \exists k \exists t_2 < \infty: wait\ to\ re_k? \text{ during } (t_2, t_2 + v_k) \\ \Rightarrow \{\text{lemma 5.9}\} & \quad \exists k \exists t_2 < \infty: no \{re_k!, re_k\} \text{ during } (t_2, t_2 + v_k) \\ \Rightarrow \{C^{P_k}\} & \quad (\exists k: error_k) \end{aligned}$$

Next we try to prove $(\exists k: error_k) \rightarrow (\exists t < \infty: wait\ to\ all\ at\ t\ until\ comm)$.

From $\exists k: error_k$ we obtain, by C^{P_k} , $\exists k \exists t_0 < \infty: no \{re_k!, re_k\} \text{ during } (t_0, t_0 + v_k)$, and thus $\exists k \exists t_0 < \infty: no\ comm\ via\ re_k \text{ during } (t_0, t_0 + v_k)$. With the current specification of W , however, nothing can be derived from this. The specification of W only expresses how W should behave if it does something on any of the channels. But then W need not do anything; even the simple program **skip** would satisfy its specification. Therefore we modify the specification for W as follows: $C_1^W \wedge C_2^W: \{time = 0\} W \{true\}$, with

$$\begin{aligned} C_1^W & \equiv \forall t_1 < \infty: wait\ to\ all\ at\ t_1\ until\ comm \rightarrow \\ & \quad \exists k \exists t_2 \leq t_1: wait\ to\ re_k? \text{ during } (t_2, t_2 + v_k), \\ C_2^W & \equiv \forall t_3 < \infty: no\ comm\ via\ re_k \text{ during } (t_3, t_3 + v_k) \rightarrow \\ & \quad \exists t_4 \leq t_3 + v_k + K: wait\ to\ all\ at\ t_4\ until\ comm. \end{aligned}$$

Note that C_0^W follows from C_2^W , because $wait\ to\ re_k? \text{ during } (t_0, t_0 + v_k)$ implies by lemma 5.9, $no \{re_k!, re_k\} \text{ during } (t_0, t_0 + v_k)$, and hence $no\ comm\ via\ re_k \text{ during } (t_0, t_0 + v_k)$. Now the proof proceeds as follows, for all k ,

$$\begin{aligned} & (\exists k: error_k) \\ \Rightarrow \{C^{P_i}\} & \quad \exists k \exists t_0 < \infty: no \{re_k!, re_k\} \text{ during } (t_0, t_0 + v_k) \\ \Rightarrow \{\text{definition}\} & \quad \exists k \exists t_0 < \infty: no\ comm\ via\ re_k \text{ during } (t_0, t_0 + v_k) \\ \Rightarrow \{C_2^W\} & \quad \exists k \exists t_0 < \infty \exists t_4 \leq t_0 + v_k + K: wait\ to\ all\ at\ t_4\ until\ comm \\ \Rightarrow \{\text{calculus}\} & \quad \exists t_4 < \infty: wait\ to\ all\ at\ t_4\ until\ comm. \end{aligned}$$

Figure 2. Implementation of the watchdog timer.

6.2 Implementing the watchdog timer

Next we design a program implementing watchdog process W that satisfies the required specification. Since W has to watch all processes $P_1, \dots, P_n$ simultaneously, our first design step is to implement W as a parallel composition, $W \equiv W_1 \parallel \dots \parallel W_n \parallel A$.

Process W_i is a watchdog for P_i , and it signals process A via channel a_i as soon as there is no communication on re_i for at least v_i time units. Process A waits for a signal on any of the a_i 's; after receipt of a signal it tries to send a message on al (see figure 2). We give specifications for W_i and A and prove that they are sufficient to derive the specification of W . The specification for W_i expresses that W_i tries to communicate via a_i only if it has been waiting to communicate via re_i during a period of v_i time units. On the other hand, if there is a period of v_i time units during which no communication via re_i occurs, then W_i will try to communicate via a_i within a certain time bound K_i . Define

$$\begin{aligned} C_1^{W_i} &\equiv \forall t_1 < \infty : \text{wait to } a_i! \text{ at } t_1 \text{ until comm} \rightarrow \\ &\quad \exists t_2 \leq t_1 : \text{wait to } re_i? \text{ during } (t_2, t_2 + v_i), \\ C_2^{W_i} &\equiv \forall t_1 < \infty : \text{no comm via } re_i \text{ during } (t_3, t_3 + v_i) \rightarrow \\ &\quad \exists t_4 \leq t_3 + v_i + K_i : \text{wait to } a_i! \text{ at } t_4 \text{ until comm.} \end{aligned}$$

Then W_i is specified by $C_1^{W_i} \wedge C_2^{W_i} : \{time = 0\} W_i \{true\}$.

The specification for A asserts that it tries to send a message via al only if there was a preceding communication via one of the a_k . If A is not waiting to communicate via one of the a_k at a certain point of time, then within, say, K_A time units it will wait to communicate via al until the actual communication can be performed. Define

$$\begin{aligned} C_1^A &\equiv \forall t_1 < \infty : \text{wait to } al! \text{ at } t_1 \text{ until comm} \rightarrow \\ &\quad \exists k \exists t_2 \leq : \text{comm via } a_k \text{ during } [t_2, t_2 + K_c), \\ C_2^A &\equiv \forall t_3 < \infty : \neg \text{wait to } a_k? \text{ at } t_3 \rightarrow \\ &\quad \exists t_4 < t_3 + K_A : \text{wait to } al! \text{ at } t_4 \text{ until comm.} \end{aligned}$$

Then $C_1^A \wedge C_2^A : \{time = 0\} A \{true\}$.

We show that $W_1 \parallel \dots \parallel W_n \parallel A$ satisfies the specification of W (using the specifications of $W_1, \dots, W_n$, and A only). By the repeated application of the simple parallel

composition rule, we obtain the conjunction of the commitments of the processes: $\bigwedge_{i=1}^n (C_1^{W_i} \wedge C_2^{W_i}) \wedge C_1^A \wedge C_2^A$. By the well-formedness axiom and the conjunction rule we can add *Well Form* $_{\{a_k, a_k!, a_k? | k=1, \dots, n\}}$, leading to the following commitment: $\bigwedge_{i=1}^n (C_1^{W_i} \wedge C_2^{W_i}) \wedge C_1^A \wedge C_2^A \wedge \text{Well Form}_{\{a_k, a_k!, a_k? | k=1, \dots, n\}}$. This implies C_1^W as follows, for all $t_1 < \infty$,

$$\begin{aligned}
& \text{wait to all! at } t_1 \text{ until comm} \\
\Rightarrow \{C_1^A\} & \quad \exists k \exists t_2 \leq t_1 : \text{comm via } a_k \text{ during } [t_2, t_2 + K_c) \\
\Rightarrow \{\text{definition}\} & \quad \exists k \exists t_2 \leq t_1 : \text{wait to } a_k! \text{ at } t_2 \text{ until comm} \\
\Rightarrow \{C_1^{W_k}\} & \quad \exists k \exists t_2 \leq t_1 \exists t_3 \leq t_2 : \text{wait to } re_k? \text{ during } (t_3, t_3 + v_k). \\
\Rightarrow \{t_3 \leq t_2 \leq t_1\} & \quad \exists k \exists t_3 \leq t_1 : \text{wait to } re_k? \text{ during } (t_3, t_3 + v_k).
\end{aligned}$$

Next we prove C_2^W . For all $t_3 < \infty$,

$$\begin{aligned}
& \text{no comm via } re_k \text{ during } (t_3, t_3 + v_k) \\
\Rightarrow \{C_2^{W_k}\} & \quad \exists t_4 \leq t_3 + v_k + K_k : \text{wait to } a_k! \text{ at } t_4 \text{ until comm} \\
\Rightarrow \{\text{lemma 5.10}\} & \quad \exists t_4 \leq t_3 + v_k + K_k \forall t_5, t_4 < t_5 < t_4 + K_c : \neg \text{wait to } a_k? \text{ at } t_5 \\
\Rightarrow \{C_2^A\} & \quad \exists t_4 \leq t_3 + v_k + K_k \forall t_5, t_4 < t_5 < t_4 + K_c \exists t_6 < t_5 + K_A : \\
& \quad \text{wait to all! at } t_6 \text{ until comm} \\
\Rightarrow \{\text{calculus}\} & \quad \exists t_4 \leq t_3 + v_k + K_k \exists t_6 \leq t_4 + K_A : \text{wait to all! at } t_6 \text{ until comm} \\
\Rightarrow \{\text{calculus}\} & \quad \exists t_6 \leq t_3 + v_k + K_k + K_A : \text{wait to all! at } t_6 \text{ until comm.}
\end{aligned}$$

Hence the specification of W can be derived provided $K_k + K_A \leq K$, for all k .

6.3 Final implementations

Finally, we give implementations for the processes A and W_i , and we show that these programs meet the required specifications.

Implementation of A: First we show that A can be implemented as $[\Box_{i=1}^n a_i? \rightarrow \text{all!}]$.

We have to prove

$$C_1^A \wedge C_2^A : \{time = 0\} [\Box_{i=1}^n a_i? \rightarrow \text{all!}] \{true\}.$$

Define, for $A \equiv [\Box_{i=1}^n a_i? \rightarrow \text{all!}]$ and $i \in \{1, \dots, n\}$,

$$\begin{aligned}
C_i^1 & \equiv \text{wait in } A \text{ during } [t_0, t) \wedge \text{comm via } a_i \text{ during } [t, t + K_c), \text{ and} \\
C_i^2 & \equiv \text{wait to all! at } t + K_c \text{ until comm.}
\end{aligned}$$

We apply the rule for guarded command without delay using

$$\begin{aligned}
C_i & \equiv \exists t, t_0 \leq t < \infty : C_i^1 \wedge C_i^2, \quad C_{\text{nonterm}} \equiv \text{wait in } A \text{ during } [t_0, \infty), \\
p_i & \equiv \exists t, t_0 \leq t < \infty : C_i^1 \wedge time = t + K_c, \text{ and } q_i \equiv true. \text{ Then}
\end{aligned}$$

1. wait in A during $[t_0, \infty) \wedge time = \infty \rightarrow C_{\text{nonterm}}$.

2. $\exists t, t_0 \leq t < \infty: \text{wait in } A \text{ during } [t_0, t) \wedge$
 $\text{comm via } a_i \text{ during } [t, t + K_c) \wedge \text{time} = t + K_c \rightarrow p_i, \text{ for } i \in \{1, \dots, n\}.$
3. $C_i: \{p_i\} \text{ all } \{q_i\}, \text{ for } i \in \{1, \dots, n\} \text{ can be derived as follows.}$

By the send axiom and the consequence rule,

$$\text{wait to all at } t_1 \text{ until comm: } \{\text{time} = t_1\} \text{ all } \{\text{true}\}.$$

Applying the adaptation rule, with $\text{exp} \equiv t_1 + K_c$, we obtain

$$(p_i[t_1 + K_c/\text{time}] \wedge \text{wait to all at } t_1 + K_c \text{ until comm}):$$

$$\{p_i[t_1 + K_c/\text{time}] \wedge \text{time} = t_1 + K_c\} \text{ all } \{\text{true}\}.$$

Since $(p_i[t_1 + K_c/\text{time}] \wedge \text{wait to all at } t_1 + K_c \text{ until comm}) \rightarrow$
 $(\exists t, t_0 \leq t < \infty: C_i^1 \wedge t_1 + K_c = t + K_c \wedge \text{wait to all at } t_1 + K_c \text{ until comm}) \rightarrow$
 $(\exists t, t_0 \leq t < \infty: C_i^1 \wedge C_i^2)$, the consequence rule leads to

$$C_i: \{p_i[t_1 + K_c/\text{time}] \wedge \text{time} = t_1 + K_c\} \text{ all } \{\text{true}\}.$$

By the substitution rule, using $\text{exp} \equiv 0$,

$$C_{\text{nonterm}}[0/t_0] \vee \bigvee_{i=1}^n C_i[0/t_0]: \{\text{time} = 0\} \equiv [\bigwedge_{i=1}^n a_i? \rightarrow \text{all}] \{\text{true}\}.$$

Since $p_i \rightarrow \exists t_1: p_i[t_1 + K_c/\text{time}] \wedge \text{time} = t_1 + K_c$, by the consequence rule,

$$C_i: \{p_i\} \text{ all } \{q_i\}.$$

Then by the rule for guarded command without delay we obtain

$$C_{\text{nonterm}} \vee \bigvee_{i=1}^n C_i: \{\text{time} = t_0\} [\bigwedge_{i=1}^n a_i? \rightarrow \text{all}] \{\text{true}\}.$$

By the substitution rule, using $\text{exp} \equiv 0$,

$$C_{\text{nonterm}}[0/t_0] \vee \bigvee_{i=1}^n C_i[0/t_0]: \{\text{time} = 0\} [\bigwedge_{i=1}^n a_i? \rightarrow \text{all}] \{\text{true}\}.$$

We prove that this commitment implies $C_1^A \wedge C_2^A$.

- First prove $C_1^A \equiv \forall t_1 < \infty: \text{wait to all at } t_1 \text{ until comm} \rightarrow$
 $\exists k \exists t_2 \leq t_1: \text{comm via } a_k \text{ during } [t_2, t_2 + K_c)$
 — By the definition of *wait in A during* $[0, \infty)$, $C_{\text{nonterm}}[0/t_0]$ leads to
 $\text{no } \{a!, \text{all}\} \text{ during } [0, \infty)$, and thus $\forall t_1 < \infty: \neg \text{wait to all at } t_1 \text{ until comm}.$
 — $\bigvee_{i=1}^n C_i[0/t_0]$ implies $\exists k \exists t < \infty: C_k^1[0/t_0]$, and thus, by the definition of *wait*
in A during $[0, t)$, $\exists k \exists t < \infty: \text{no } \{a!, \text{all}\} \text{ during } [0, t) \wedge \text{comm via } a_k \text{ during } [t, t + K_c).$
 Thus, for all $t_1 < \infty$, *wait to all at } t_1 \text{ until comm}* implies $t_1 \geq t$, and hence
 $\exists t \leq t_1: \text{comm via } a_k \text{ during } [t, t + K_c).$
- Next we prove C_2^A , that is,
 $\forall t_3 < \infty: \neg \text{wait to } a_k? \text{ at } t_3 \rightarrow \exists t_4 < t_3 + K_A: \text{wait to all at } t_4 \text{ until comm}$
 — From $C_{\text{nonterm}}[0/t_0]$, we obtain $\forall i \in \{1, \dots, n\}: \text{wait to } a_i? \text{ during } [0, \infty)$,
 and hence $\forall t_3 < \infty \forall k \in \{1, \dots, n\}: \text{wait to } a_k? \text{ at } t_3.$
 — Assume $\neg \text{wait to } a_k? \text{ at } t_3$, for $t_3 < \infty$. By $C_k[0/t_0]$, there exists a $t < \infty$ such
 that *wait to } a_k? \text{ during } [0, t) and *wait to all at } t + K_c \text{ until comm}.* Then $t \leq t_3$,
 thus $t + K_c \leq t_3 + K_c$, and hence $\exists t_4 \leq t_3 + K_A: \text{wait to all at } t_4 \text{ until comm}.$ This
 leads to C_2^A , provided $K_c < K_A$.*

Implementation of W_i : Next we implement W_i by $*[re_i? \rightarrow \text{skip} \square \text{delay } v_i \rightarrow a_i!]$ and show that, under certain restrictions, this program satisfies the required specification $C_1^{W_i} \wedge C_2^{W_i} : \{time = 0\} W_i \{true\}$. Define

$$\begin{aligned} C_1(t_1) &\equiv \text{wait to } a_i! \text{ at } t_1 \text{ until comm} \rightarrow \\ &\quad \exists t_2 \leq t_1 : \text{wait to } re_i? \text{ during } (t_2, t_2 + v_i), \\ C_2(t_3) &\equiv \text{no comm via } re_i \text{ during } (t_3, t_3 + v_i) \rightarrow \\ &\quad \exists t_4 \leq t_3 + v_i + K_i : \text{wait to } a_i! \text{ at } t_4 \text{ until comm.} \end{aligned}$$

Then

$$C_1^{W_i} \equiv \forall t_1 < \infty : C_1(t_1) \text{ and } C_2^{W_i} \equiv \forall t_3 < \infty : C_2(t_3).$$

We apply the iteration rule with

$$C_{\text{nonterm}} \equiv C_1^{W_i} \wedge C_2^{W_i}, \text{ and } C \equiv \forall t_1 < time : C_1(t_1) \wedge \forall t_3 < time : C_2(t_3).$$

If the assumptions for the iteration rule are fulfilled – which is shown below – we obtain

$$C_{\text{nonterm}} \wedge time = \infty : \{C\} * [re_i? \rightarrow \text{skip} \square \text{delay } v_i \rightarrow a_i!] \{false\}.$$

Since $C_{\text{nonterm}} \wedge time = \infty \rightarrow C_1^{W_i} \wedge C_2^{W_i}$, $time = 0 \rightarrow C$, and $false \rightarrow true$, the consequence rule leads to $C_1^{W_i} \wedge C_2^{W_i} : \{time = 0\} W_i \{true\}$.

To apply the iteration rule we have to prove

$$(\forall t < \infty \exists t_0 > t : C[t_0/time]) \rightarrow C_{\text{nonterm}}, \quad (5)$$

$$C : \{C\} [re_i? \rightarrow \text{skip} \square \text{delay } v_i \rightarrow a_i!] \{C\}. \quad (6)$$

Proof of (5). Observe that $(\forall t < \infty \exists t_0 > t : C[t_0/time]) \equiv$

$$\begin{aligned} &(\forall t < \infty \exists t_0 > t : (\forall t_1 < t_0 : C_1(t_1) \wedge \forall t_3 < t_0 : C_2(t_3))) \rightarrow \\ &(\forall t_1 < \infty : C_1(t_1) \wedge \forall t_3 < \infty : C_2(t_3)) \equiv C_{\text{nonterm}}, \text{ and hence (5) holds.} \end{aligned}$$

Proof of (6). Consider $C^\alpha \equiv \forall t_1 < t_5 : C_1(t_1) \wedge \forall t_3 < t_5 : C_2(t_3)$, and

$$C^\beta \equiv \forall t_1, t_5 \leq t_1 < time : C_1(t_1) \wedge \forall t_3, t_5 \leq t_3 < time : C_2(t_3). \text{ Then } C \leftrightarrow C^\alpha \wedge C^\beta.$$

Below we derive

$$C^\beta : \{time = t_5\} [re_i? \rightarrow \text{skip} \square \text{delay } v_i \rightarrow a_i!] \{C^\beta\}. \quad (7)$$

From (7) we obtain by the adaptation rule, with $p \equiv C^\alpha$ and $exp \equiv t_5$,

$$C^\alpha \wedge C^\beta : \{C^\alpha \wedge time = t_5\} [re_i? \rightarrow \text{skip} \square \text{delay } v_i \rightarrow a_i!] \{C^\alpha \wedge C^\beta\}.$$

Using $C \leftrightarrow (C^\alpha \wedge C^\beta)$, by the consequence rule,

$$C : \{C^\alpha \wedge time = t_5\} [re_i? \rightarrow \text{skip} \square \text{delay } v_i \rightarrow a_i!] \{C\}.$$

Since $C \rightarrow (\exists t_5 : C[t_5/time] \wedge time = t_5) \rightarrow (\exists t_5 : C^\alpha \wedge time = t_5)$, the quantification rule and the consequence rule lead to (6).

Proof of (7). Let $G \equiv [re_i? \rightarrow \mathbf{skip} \square \mathbf{delay} \ v_i \rightarrow a_i!]$.

Apply the rule for guarded command with delay, using

$$p_1 \equiv \exists t, t_5 \leq t < t_5 + v_i: \text{wait to } re_i? \text{ during } [t_5, t) \wedge \text{comm via } re_i \text{ during } [t, t + K_c) \wedge \\ \text{no } \{a_i, a_i!\} \text{ during } [t_5, t + K_c) \wedge \text{time} = t + K_c.$$

Then $\exists t, t_5 \leq t < t_5 + v_i: \text{wait in } G \text{ during } [t_5, t) \wedge \text{comm } re_i \text{ in } G \text{ from } t \rightarrow p_1$, thus we can derive (7), provided

$$C^\beta: \{p_1\} \mathbf{skip} \{C^\beta\}, \quad (8)$$

$$C^\beta: \{\text{wait in } G \text{ during } [t_5, t_5 + v_i) \wedge \text{time} = t_5 + v_i\} a_i! \{C^\beta\}. \quad (9)$$

Proof of (8). Derive by the skip axiom $\text{time} = t_0: \{\text{time} = t_0\} \mathbf{skip} \{\text{time} = t_0\}$.

Then by the adaptation rule, with $\text{exp} \equiv t + K_c$ and

$$p \equiv t_5 \leq t < t_5 + v_i \wedge \text{wait to } re_i? \text{ during } [t_5, t) \wedge \text{comm via } re_i \text{ during } [t, t + K_c) \wedge \\ \text{no } \{a_i, a_i!\} \text{ during } [t_5, t + K_c),$$

we obtain $p \wedge \text{time} = t + K_c: \{p \wedge \text{time} = t + K_c\} \mathbf{skip} \{p \wedge \text{time} = t + K_c\}$.

By the consequence rule and the quantification rule we obtain $p_1: \{p_1\} \mathbf{skip} \{p_1\}$. Observe that

$$\begin{aligned} p_1 &\Rightarrow \forall t_1, t_5 \leq t_1 < \text{time}: \neg \text{wait to } a_i! \text{ at } t_1 \wedge \neg \text{comm via } a_i \text{ at } t_1 \\ &\Rightarrow \forall t_1, t_5 \leq t_1 < \text{time}: \neg \text{wait to } a_i! \text{ at } t_1 \text{ until comm} \\ &\Rightarrow \forall t_1, t_5 \leq t_1 < \text{time}: C_1(t_1), \end{aligned}$$

and

$$\begin{aligned} p_1 &\Rightarrow \exists t, t_5 \leq t < t_5 + v_i: \text{comm via } re_i \text{ during } [t, t + K_c) \wedge \text{time} = t + K_c \\ &\Rightarrow \forall t_3, t_5 \leq t_3 < \text{time} \exists t, t_3 \leq t < t_3 + v_i: \text{comm via } re_i \text{ at } t \\ &\Rightarrow \forall t_3, t_5 \leq t_3 < \text{time}: \neg \text{no comm via } re_i \text{ during } (t_3, t_3 + v_i) \\ &\Rightarrow \forall t_3, t_5 \leq t_3 < \text{time}: C_2(t_3). \end{aligned}$$

Hence $p_1 \rightarrow C^\beta$, and thus the consequence rule leads to (8).

Proof of (9). Define

$$C^a \equiv \exists t \geq t_5 + v_i: \text{wait to } a_i! \text{ at } t_5 + v_i \text{ until comm at } t \wedge \text{time} = t + K_c.$$

From the send rule, the consequence rule and the substitution rule (replacing t_0 by $t_5 + v_i$) we obtain $C^a: \{\text{time} = t_5 + v_i\} a_i! \{C^a \wedge \text{time} < \infty\}$.

Define $C^b \equiv \text{wait to } re_i? \text{ during } [t_5, t_5 + v_i) \wedge \text{no } \{a_i, a_i!\} \text{ during } [t_5, t_5 + v_i)$.

Then by the adaptation rule we can derive

$$C^a \wedge C^b: \{\text{time} = t_5 + v_i \wedge C^b\} a_i! \{C^a \wedge C^b \wedge \text{time} < \infty\}.$$

Since $\text{wait in } G \text{ during } [t_5, t_5 + v_i) \wedge \text{time} = t_5 + v_i \rightarrow \text{time} = t_5 + v_i \wedge C^b$, we obtain (9) by the consequence rule, if $C^a \wedge C^b$ implies C^β . Recall that

$$\begin{aligned} C^\beta &\equiv (\forall t_1, t_5 \leq t_1 < \text{time}: \text{wait to } a_i! \text{ at } t_1 \text{ until comm} \rightarrow \\ &\quad \exists t_2 \leq t_1: \text{wait to } re_i? \text{ during } (t_2, t_2 + v_i)) \wedge \\ &\quad (\forall t_3, t_5 \leq t_3 < \text{time}: \text{no comm via } re_i \text{ during } (t_3, t_3 + v_i)) \rightarrow \\ &\quad \exists t_4 \leq t_3 + v_i + K_i: \text{wait to } a_i! \text{ at } t_4 \text{ until comm}). \end{aligned}$$

It remains to prove $C^a \wedge C^b \rightarrow C^b$:

- For all $t_1, t_5 \leq t_1 < \text{time}$: if wait to $a_i!$ at t_1 until comm, then from $C^a \wedge C^b$, $t_1 \geq t_5 + v_i$. Hence, from C^b , there exists a $t_2 < t_1$ (viz., t_5) such that wait to $re_i?$ during $(t_2, t_2 + v_i)$, provided $v_i > 0$.
- Assume, for all $t_3, t_5 \leq t_3 < \text{time}$: no comm via re_i during $(t_3, t_3 + v_i)$. From C^a we obtain wait to $a_i!$ at $t_5 + v_i$ until comm). Hence, there exists a $t_4 \leq t_3 + v_i + K_i$ such that wait to $a_i!$ at t_4 until comm) if $t_5 + v_i \leq t_3 + v_i + K_i$. Since $t_5 \leq t_3$, this holds provided $K_i \geq 0$.

Conclusion: By giving programs that implement A and W_i , we have obtained an implementation that satisfies the top-level specification for a watchdog timer as given in §6.1. To conclude this example, we analyse the requirements which have been imposed upon the constants K, K_i, K_A and v_i to prove the correctness of our implementation. To justify the refinement step from the previous section, we required $K_i + K_A \leq K$, for all $i = 1, \dots, n$. The implementation given in this section has been proved to satisfy the specification for all K_A and K_i such that $K_i < K_A$ and $K_i \geq 0$, and provided $v_i > 0$, for all $i = 1, \dots, n$. Observe that if $K > K_c$ then for $K_A = K$ and $K_i = 0$ we have $K_c < K_A$ and $K_i \geq 0$, and also $K_i + K_A = K \leq K$. Hence, if $v_i > 0$ for all $i = 1, \dots, n$, and $K > K_c$, i.e., the constant in the specification must be greater than the duration of a communication, then our implementation meets the top-level specification for W .

7. Adding assumptions to real-time

In general, real-time embedded systems have an intensive interaction with their environment, and usually their correctness strongly depends on assumptions about the environment. Therefore it is convenient to use correctness formulae in which these assumptions can be expressed. Hence, similar to §4, we extend the formulae $C:\{p\}S\{q\}$ from §5 with a fourth assertion, called *assumption*, leading to formulae of the form $(A, C):\{p\}S\{q\}$. Assumption A should neither contain program variables nor the special variable *time*.

In this assumption/commitment formalism for real-time properties, we can now express, for instance, in the assumption when the environment of a process is waiting to communicate. With such an assumption we can determine when the communication must take place. For example,

$$(A \equiv \text{wait to } c! \text{ at } 5 \text{ until comm} \wedge \text{no comm via } c \text{ during } [3, 5),$$

$$C \equiv \text{comm via } c \text{ during } [5, 5 + K_c);$$

$$\{time = 3\}c?\{time = 5 + K_c\}.$$

Note that, by using the maximal parallelism model, a communication takes place as soon as both process and environment are ready to perform the communication.

In the remainder of this section we indicate how the proof system from §5 can be extended to obtain a compositional proof system for these assumption/commitment-formulae. First we consider a compositional rule for the parallel composition of S_1

and S_2 . Concerning pre-conditions, post-conditions and commitments, the rule is identical to rule 5.27 for the commitment-formalism. For the assumptions, we have same requirements as in rule 4.2: $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$.

Rule 7.1. (Parallel composition)

$$\frac{\begin{array}{l} (A_1, C_1): \{p_1\} S_1 \{q_1\}, (A_2, C_2): \{p_2\} S_2 \{q_2\} \\ q_1[t_1/time] \wedge q_2[t_2/time] \wedge time = \max(t_1, t_2) \rightarrow q \\ C_1[t_1/time] \wedge C_2[t_2/time] \wedge time = \max(t_1, t_2) \rightarrow C \\ A \wedge C_1 \rightarrow A_2, A \wedge C_2 \rightarrow A_1 \end{array}}{(A, C): \{p_1 \wedge p_2\} S_1 \parallel S_2 \{q\}}.$$

provided t_1 and t_2 are fresh logical variables, and $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$. A typical application of this rule can be found in the next example.

Example 7.1. Consider the following specifications (**delay** d is used to represent any internal actions which takes d time units and assume communications take one time unit, i.e., $K_c = 1$):

$$\begin{aligned} (A_1 \equiv & \text{wait to } c? \text{ at } 2 \text{ until comm} \wedge \text{no comm via } c \text{ during } [0, 2) \wedge \\ & \text{wait to } d? \text{ at } 6 \text{ until comm} \wedge \text{no comm via } d \text{ during } [3, 6), \\ (C_1 \equiv & \text{comm via } c \text{ during } [2, 3) \wedge \text{wait to } d! \text{ at } 3 \text{ until comm} \wedge \\ & \text{no comm via } d \text{ during } [0, 3)): \\ & \{time = 0\} c!; d!; \text{delay } 2 \{time = 9\}, \text{ and} \\ (A_2 \equiv & \text{wait to } d! \text{ at } 3 \text{ until comm} \wedge \text{no comm via } d \text{ during } [0, 3), \\ C_2 \equiv & \text{wait to } d? \text{ at } 6 \text{ until comm} \wedge \text{no comm via } d \text{ during } [0, 6) \wedge \\ & \text{comm via } d \text{ during } [6, 7)): \\ & \{time = 0\} \text{delay } 6; d? \{time = 7\}. \end{aligned}$$

Take for $(c!; d!; \text{delay } 2) \parallel (\text{delay } 6; d?)$ the following assumption:

$$A \equiv \text{wait to } c? \text{ at } 2 \text{ until comm} \wedge \text{no comm via } c \text{ during } [0, 2).$$

Since $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$, the parallel composition rule leads to

$$(A, C_1 \wedge C_2): \{time = 0\} (c!; d!; \text{delay } 2) \parallel (\text{delay } 6; d?) \{time = 9\}.$$

Using a consequence rule we can easily derive from $C_1 \wedge C_2$ the following commitment: $\text{comm via } c \text{ during } [2, 3) \wedge \text{comm via } d \text{ during } [6, 7)$. $\square$

Similar to section 4, to achieve a sound rule for parallel composition, an inductive relation between assumption and commitment is necessary to avoid circularity. In our real-time specifications we require for the validity of $(A, C): \{p\} S \{q\}$ that there exists a $\delta > 0$ such that

1. for all t with $0 \leq t < \delta$: C holds at t , and
2. for all $t \geq \delta$: if A holds at $t - \delta$ then C holds at t .

In our examples this requirement is fulfilled if *comm via D during* $[t_0, t_1)$ is not considered as an abbreviation but as a primitive which is trivially true at all points of time before t_1 .

Other rules and axioms for the assumption/commitment formalism can be obtained by adapting the commitment-based proof system of the previous section. We simply add assumption *true* to the rule and axioms for atomic statements, and the proof system is extended by a rule that allows the addition of an assumption to strengthen commitment and post-condition. To formulate this rule, let $p@t$ denote assertion p at time t , ignoring the part of p that refers to point of time after t . (By definition, $(p@t) \equiv \text{true}$ if $t < 0$.) Then, for all $\delta > 0$, we have the following rule.

Rule 7.2. (Strengthen)

$$\frac{\begin{array}{l} (A_1, C_1): \{p_1\} S \{q_1\} \\ \forall t: (A @ (t - \delta)) \wedge (C_1 @ t) \rightarrow (C @ t) \\ \forall t: (A @ (t - \delta)) \wedge (q_1 @ t) \rightarrow (q @ t) \end{array}}{(A_1 \wedge A, C): \{p_1\} S \{p_1\}}.$$

Furthermore, the rules for compound statements have to be adapted, and in the consequence rule we require that all implications hold point-wise.

Rule 7.3. (Consequence)

$$\frac{\begin{array}{l} (A_1, C_1): \{p_1\} S \{q_1\} \\ \forall t: (A @ t) \rightarrow (A_1 @ t), \forall t: (p @ t) \wedge (\text{time} < \infty) \rightarrow (p_1 @ t) \\ \forall t: (C_1 @ t) \rightarrow (C @ t), \forall t: (q_1 @ t) \rightarrow (q @ t) \end{array}}{(A, C): \{p\} S \{q\}}.$$

8. Related work and state of the art

For concurrent programs communicating via message passing as well as for shared variable concurrency, one can observe a development from non-compositional proof methods which require the (final) program text for their application, such as Owicki & Gries (1976), Apt *et al* (1980) and Levin & Gries (1981), towards compositional theories, e.g. Chen & Hoare (1981, pp. 1–12), Soundararajan (1984b), Stirling (1986, pp. 407–415), Zwiers (1989) and Stølen (1990) (see de Roever 1985b, pp. 181–207, and Hooman *et al* 1986 for an overview of this development). An early Indian pioneer in compositional proof methods for concurrency is Soundararajan (Soundararajan 1984; Sobel & Soundararajan 1985, pp. 343–359). Whereas these methods verify only safety properties, with temporal logic (Pnueli 1977, pp. 46–57; Manna & Pnueli 1982, pp. 163–255) also liveness (progress) properties can be verified. Compositional proof systems for temporal logic have been given in Barringer *et al* (1984, pp. 51–63) and Nyugen *et al* (1986). In Pandya & Joseph (1991) a compositional proof system called P-A logic (for presupposition–affirmation logic) is described for establishing weak total correctness and weak divergence correctness of CSP-like distributed programs with synchronous and asynchronous communication. This extension allows compositional deadlock proofs and, moreover, compositional proof rules are given for *until*-properties of the form $Q \text{ until } R$, where Q and R are assertions over communication traces. It seems

that this paper describes how far one can go towards proving liveness in a compositional framework using a non-temporal formalism in an assumption-commitment based setting.

Interestingly, more involved programming language fragments for concurrency, such as the concurrency fragment of Ada (1983) or those for monitor based languages, have not been characterized until now through compositional trace-based methods, although their non-compositional characterization was possible and has been given – see de Roever (1985a, pp. 213–260) for an overview. However, if one allows locations inside programs as observables, there exists a straightforward technique to convert non-compositional proof methods for concurrency to compositional ones, as reported in Gerth & de Roever (1986). Similarly, Stirling (1986, pp. 407–416) reports how a basically non-compositional proof method for shared variable concurrency, such as the one by Owicki & Gries (1976), can be reformulated compositionally within a framework based on relevance logic. Also for the parallel object oriented language POOL first non-compositional proof methods (America & de Boer 1990) have been developed based on the method of Apt *et al* (1980). The principal author, de Boer, recently reformulated his proof system along history-based compositional lines (de Boer 1991) using the work of Zwiers (1989) as a starting point.

In the present paper we discuss a compositional proof system for distributed message passing in which assumptions can be made about the behaviour of the environment in the style of Misra & Chandy (1981) and Zwiers *et al* (1984). The main idea of the method is that suitable assumptions about the environment reduce the immense number of possible behaviours of complex reactive systems. Misra & Chandy (1981) were the first ones to demonstrate the advantages of assumptions in the hierarchical design and verification of distributed processes with message passing. They proposed a compositional rule for the parallel operator and demonstrated their method on several examples. These ideas have been formalized by Zwiers *et al* (1984), resulting in a compositional proof system for assumption/commitment based specifications together with its soundness and completeness proof. The examples in Ossefort (1983) show that the Misra–Chandy method is easy to use indeed and that it leads to simple and natural correctness proofs. In Pandya (1988) and Pandya & Joseph (1991), the formalism of Zwiers *et al* (1984) is extended to asynchronous communication and progress properties. Also related is the formalism given by Stark (1985, pp. 369–391), who uses rely- and guarantee-conditions for deriving global liveness properties of a distributed system. Interestingly, at present new non-standard applications of the assumption-commitment framework are burgeoning, e.g., for characterizing specifications of fault-tolerant processes or within development methods for mutual exclusion algorithms in which the final algorithm is obtained by a series of error containing approximations in which errors are gradually removed until all are absent, see e.g. Cau & Kuiper (1991). A compositional theory for action refinement is developed by Janssen *et al* (1991), in a setting of partial orders, and this theory is applied to proving the correctness of distributed databases by formalizing the notion of serializability. It contains an example of how a general specification using an unbounded number of processors is refined to an implementation using two processors. Recently, a number of developments in assumption/commitment based reasoning have taken place; for the state of the art consult the papers by Pandya (1989, pp. 622–640) and Abadi & Lamport (1989, pp. 1–41).

The present paper focusses on concurrent processes with synchronous message passing along channels. What about compositional approaches to shared-variable

concurrency and asynchronous message passing? As to shared-variable concurrency, a basic observation is made by Aczel (as reported in de Roever 1985b, pp. 181–207), in which in the semantics of a process a distinction is made between a so-called “component action” Π (an action of the process itself) and an “environment action” E (an update to a shared variable by another process). Then (x, Π) and (x, E) are the analogues of the communication records in the channel-based theory above. The original reference in which these notions were informally introduced is Jones (1981); Jones (1983) is a more accessible reference to these ideas and contains a proposal for assumption/commitment based reasoning (called *rely/guarantee reasoning*) about shared variable concurrency together with the formulation of a compositional proof rule for the parallel operator. The idea is exemplified in Woodcock & Dickinson (1988) and formally worked out in Stølen (1990). In a mixed (temporal logic)-(transition system) based approach, asynchronous communication is characterized compositionally in the work by Jonsson (1987a, 1987b, pp. 152–166). It is shown (de Boer *et al* 1990) that for a compositional description of any programming language based upon asynchronous communication a trace model is sufficient, i.e., no additional structures to encode some relevant branching information (trees, failure sets) are needed. A compositional axiomatization is given (Hooman *et al* 1990, pp. 242–261, 1992) for the graphical specification language Statecharts which includes features like concurrency, broadcast communication, and time-out.

A dichotomy is observed (Zwiers & Roever 1989) in compositional proof theories for concurrency; one class of methods (including, e.g., temporal logic and VDM), is based on programs as predicates, and has a simple proof theory (due to the power of the consequence rule), but has trouble in characterizing sequential composition and iteration. Methods in the other class (including weakest pre-condition calculi, Hoare triples and dynamic logics), are based on programs as predicate transformers, and have no trouble in dealing with sequential composition and iteration, but are more complicated due to awkward implication rules. An attempt at unification is made (Zwiers & de Roever 1989) by using adjoints.

Except for Statecharts, these methods are not designed to verify and specify real-time properties. Now an obvious approach towards a verification theory for real-time programs is to adapt and extend an already existing method which does not incorporate any notion of time. For instance, in traditional linear temporal logic, safety and liveness properties are expressed by means of a qualitative notion of time (e.g. “eventually”, “henceforth”, “until”). In order to express real-time constraints, extensions of this logic have been proposed (Bernstein & Harter 1981, pp. 1–11; Shankar & Lam 1987; Koymans 1989) which also includes a quantitative notion of time (e.g. “eventually within 5 time units”, “always after 7 time units”). These extensions have been applied to the specification of real-time communication properties of a transmission medium (Koymans *et al* 1983, pp. 187–197) and the verification of local area network protocols (Shasha 1984, pp. 54–65). A compositional proof theory for real-time distributed message passing using an assertion language based on real-time temporal logic has been given in Hooman & Widom (1989, pp. 424–441). In Hooman (1991b) this compositional method is extended to uniprocessor implementations and priorities. Non-compositional proof methods, based on Manna & Pnueli’s (1982, pp. 163–255) classical approach to linear time temporal logic, can be found in Harel (1988) and Ostroff (1989). They express real-time properties in explicit clock temporal logic and give decision procedures for this logic.

Similarly, real-time extensions have been formulated for other methods. There is

an early paper of Haase (1981) in which time is introduced by a special variable in the weakest pre-condition calculus. Bernstein (1987) discusses several ways of modelling message passing with time-out in the non-compositional framework of Levin & Gries (1981). Zwarico & Lee (1985, pp. 169–177) have adapted Hoare's (1985) trace model (with one invariant and a satisfaction relation) to real-time. Nested parallelism is not allowed in their programming language, a restricted version of sequential composition is used, and there is no explicit mechanism for expressing time constraints. A real-time logic to analyze safety properties is defined (Jahanian & Mok 1986) based on a function which assigns a time value to each occurrence of an event. Real-time properties of sliding window protocols are verified by Shankar & Lam (1987) using special state variables, called timers, to measure the passage of time. The compositional proof system from Davies & Schneider (1989, pp. 129–159) and Schneider (1990) for timed CSP supports semantic reasoning in the framework of Reed & Roscoe (1986, pp. 314–323). Furthermore, Schneider (1990) defines a notion of time-wise refinement to transfer properties of non-timed CSP programs to their timed version, thus exploiting the hierarchy of timed and untimed models from Reed (1989, pp. 80–128). Baeten & Bergstra (1990) have incorporated real-time aspects in the process algebra of Bergstra & Klop (1984) by adding time stamps to atomic actions. In their approach atomic actions have a positive duration, whereas in the process algebra of Nicollin *et al* (1990, pp. 402–429) actions have no duration in general, except a distinguished time action which models the ticks of a synchronized global clock. Furthermore, Nicollin *et al* (1990, pp. 402–429), present a systematic approach to delay-constructs. Milner's (1989) CCS is extended by Moller & Tofts (1990, pp. 401–415) and Yi (1990, pp. 502–520) with explicit time. To obtain a calculus for shared resources, in Gerber & Lee (1990, pp. 263–277) a priority-based process algebra is presented.

This work was partially supported by Esprit-BRA project 3096: Formal Methods and Tools for the Development of Distributed and Real-Time Systems (SPEC).

References

- Ada 1983 *The Programming Language Ada* (Reference manual)
- America P, de Boer F 1990 A proof system for process-creation. In *TC-2 Working Conference on Programming Concepts and Methods*
- Apt K R, Francez N, de Roever W P 1980 *A proof system for Communicating Sequential Process*. *ACM Trans. Program. Languages Syst.* 2: 359–385
- Abadi M, Lamport L 1989 Composing specifications. In *Stepwise refinement of distributed systems* (Berlin: Springer-Verlag)
- Baeten J C M, Bergstra J A 1990 Real time process algebra, Technical Report P8916b, University of Amsterdam
- Bernstein A J 1987 Predicate transfer and timeout in message passing. *Inf. Process. Lett.* 24: 43–52
- Bernstein A, Harter P K Jr 1981 Proving real-time properties of programs with temporal logic. In *Proceedings of the 8th Annual ACM Symposium on Operating System Principles* (New York: ACM Press)
- Bergstra J A, Klop J W 1984 Process algebra for synchronous communication. *Info. Control* 60: 109–137

- Barringer H, Kuiper R, Pnueli A 1984 Now you may compose temporal logic specifications. In *Proceedings of the 16th Annual Symposium on Theory of Computing* (New York: ACM Press)
- Chen Z C, Hoare C A R 1981 Partial correctness of Communicating Sequential Processes. In *Proceedings of the IEEE International Conference on Distributed Computing Systems* (New York: IEEE Press)
- Cau A, Kuiper R 1991 Formalising Dijkstra's development strategy using Stark's formalism, Technical Report, Christian-Albrechts-University, Kiel
- de Boer F 1991 A compositional proof system for dynamic process creation. In *Proceedings Symposium on Logic in Computer Science*
- de Boer F, Kok J, Palamidessi C, Rutten J 1990 The failure of failures: towards a paradigm for asynchronous communication, Technical Report, University of Utrecht
- de Roever W P 1985a The cooperation test: a syntax directed verification method. In *Logics and Models of Concurrent Systems. NATO ASI Series F, Vol. 13* (Berlin: Springer-Verlag)
- de Roever W P 1985b The quest for compositionality – a survey of assertion-based proof systems for concurrent programs, Part I: concurrency based on shared variables. In *Proceedings of the IFIP Working Conference 1985: The role of abstract models in computer science* (Amsterdam: North-Holland)
- Davies J, Schneider S 1989 Factorizing proofs in timed CSP. In *Mathematical foundations of programming semantics. Lecture Notes in Computer Science. Vol. 442* (Berlin: Springer-Verlag)
- Gerber R, Lee I 1990 CCSR: a calculus for communicating shared resources. In *CONCOUR'90. Lecture Notes in Computer Science. Vol. 458* (Berlin: Springer-Verlag)
- Gerth R, de Roever W P 1986 Proving monitors revisited: a first step towards verifying object oriented systems. *Fundam. Inf.* 9: 371–400
- Haase V H 1981 Real-time behaviour of programs. *IEEE Trans. Software Eng.* SE-7: 494–501
- Harel E 1988 *Temporal analysis of real-time systems*, Master's thesis, The Weizmann Institute of Science, Rehovot, Israel
- Hooman J, de Roever W P 1986 The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current trends in concurrency. Lecture Notes in Computer Science. Vol. 224* (Berlin: Springer-Verlag)
- Huizing C, Gerth R, de Roever W P 1987 Full abstraction of a real-time denotational semantics for an OCCAM-like language. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages* (New York: ACM Press)
- Hoare C A R 1969 An axiomatic basis for computer programming. *Commun. ACM* 12: 576–580, 583
- Hoare C A R 1985 *Communication sequential process* (Englewood Cliffs, NJ: Prentice Hall)
- Hooman J 1991 A denotational real-time semantics for shared processors. In *Parallel Architectures and Languages Europe. Lecture Notes in Computer Science. Vol. 506* (Berlin: Springer-Verlag)
- Hooman J 1991 *Specification and compositional verification of real-time systems*, Ph D thesis, Eindhoven University of Technology. Also in (1991) *Lecture Notes in Computer Science. Vol. 558* (Berlin: Springer-Verlag)
- Harel D, Pnueli A 1985 On the development of reactive systems. In *Logics and models of concurrent systems, NATO, ASI-13* (Berlin: Springer-Verlag)
- Hooman J, Ramesh S, de Roever W P 1991 A compositional axiomatization of Statecharts. *Theor. Comput. Sci.* (to appear)
- Hooman J, Ramesh S, de Roever W P 1990 A compositional axiomatisation of safety and liveness properties of Statecharts. In *Semantics for concurrency. Workshop in Computing* (Leicester: Springer-Verlag)
- Hooman J, Widom J 1989 A temporal-logic based compositional proof system for real-time message passing. In *Parallel Architectures and Languages Europe. Lecture Notes in Computer Science. Vol. 366* (Berlin: Springer-Verlag) vol. 2
- Joseph M, Goswami A 1989 Relating computation and time, Research Report RR 138, Department of Computer Science, University of Warwick
- Jahanian F, Mok A 1986 Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng.* SE-12: 890–904
- Jones C B 1981 *Development methods for computer programs including a notion of interference*, Ph D thesis, Oxford University, Technical Monograph

- Jones C B 1983 Tentative steps towards a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* 5: 596–619
- Jonsson B 1987a *Compositional verification of distributed systems*, Ph D thesis, Uppsala University
- Jonsson B 1987b Modular verification of asynchronous networks. In *Proceedings 6th ACM Symp. Principles Distrib. Computing* (New York: ACM Press)
- Janssen W, Poel M, Zwiers J 1991 Action systems and action refinement in the development of parallel systems, Technical Report, Twente University
- Kleene S C 1952 *Introduction to metamathematics* (New York: Van Nostrand)
- Koymans R 1989 *Specifying message passing and time-critical systems with temporal logic*, Ph D thesis, Eindhoven University of Technology
- Koymans R 1990 Specifying real-time properties with metric temporal logic *Real-time Syst.* 2: 255–299
- Koymans R, Shyamasundar R K, de Roever W P, Gerth R, Arun-Kumar S 1988 Compositional semantics for real-time distributed computing. *Inf. Comput.* 79(3): 210–256
- Koymans R, Vytopyl J, de Roever W P 1983 Real-time programming and asynchronous message passing. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (New York: ACM Press)
- Levin G M, Gries D 1981 A proof technique for communicating sequential processes. *Acta Inf.* 15: 281–302
- Misra J, Chandy K M 1981 Proofs of networks of processes. *IEEE Trans. Software Eng.* SE-7: 417–426
- Milner R 1989 *Communication and concurrency* (Englewood Cliffs, NJ: Prentice Hall)
- Manna Z, Pnueli A 1982 Verification of concurrent programs: a temporal proof system. In *Foundations of Computer Science IV, Distributed Systems, Part 2-Vol. 159. Mathematical Centre Tracts* (Amsterdam: CWI)
- Moller F, Tofts C 1990 A temporal calculus of communicating systems. In *CONCUR' 90. Lecture Notes in Computer Science. Vol. 458* (Berlin: Springer-Verlag)
- Nguyen V, Demers A, Gries D, Owicki S 1986 A model and temporal proof system for networks of processes. *Distrib. Comput.* 1: 7–25
- Nicollin X, Richier J L, Sifakis J, Voiron J 1990 ATP: an algebra for timed processes. In *Proceedings IFIP Working Group Conference on Programming Concepts and Methods*
- OCCAM 1988 OCCAM2 Reference Manual, INMOS Limited (New York: Prentice-Hall)
- Owicki S, Gries D 1976 An axiomatic proof technique for parallel programs. *Acta Inf.* 6: 319–340
- Ossefort M 1983 Correctness proofs of communicating processes: Three illustrative examples from the literature. *ACM Trans. Program. Lang. Syst.* 5: 620–640
- Ostroff J 1989 *Temporal logic for real-time systems. Advanced Software Development Series* (New York: John Wiley & Sons)
- Pandya P 1988 *Compositional verification of distributed programs*, Technical Report CS-88/3 (Ph D thesis), Tata Institute of Fundamental Research, Bombay
- Pandya P 1989 Some comments on the assumption-commitment framework for compositional verification of distributed programs. In *Stepwise refinement of distributed systems. Lecture Notes in Computer Science. Vol. 430* (Berlin: Springer-Verlag)
- Pandya P, Joseph M 1991 P-A logic – a compositional proof system for distributed programs. *Distrib. Comput.* 4(4): 37–54
- Pnueli A 1977 The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*
- Reed G M 1989 A hierarchy of domains for real-time distributed computing. In *Mathematical Foundations of Programming Semantics. Lecture Notes in Computer Science. Vol. 442* (Berlin: Springer-Verlag)
- Reed G, Roscoe A 1986 A timed model for Communicating Sequential Processes. In *Proceedings of ICALP'86 Lecture Notes in Computer Science. Vol. 226* (Berlin: Springer-Verlag)
- Schneider S 1990 *Correctness and communication in real-time systems*, Ph D thesis, Oxford University
- Shanker A U, Lam S S 1987 Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distrib. Comput.* 2: 61–79
- Soundararajan N 1984 *Axiomatic semantics of communicating sequential processes. ACM Trans. Program. Lang. Syst.* 6: 647–662

- Soundararajan N 1984 A proof technique for parallel programs. *Theor. Comput. Sci.* 31: 13–29
- Shasha D E, Pnueli A, Ewald W 1984 Temporal verification of carrier-sense local area network protocols. In *Proceedings 11th ACM Symposium on Principles of Programming Languages*
- Sobel N, Soundararajan N 1985 A proof system for distributed process. In *Logics of programs. Lecture Notes in Computer Science. Vol. 193* (Berlin: Springer-Verlag)
- Stark E 1985 A proof technique for rely/guarantee properties. In *Proceedings 5th Conference on Foundations of Software Technology and Theoretical Computer Science. Lecture Notes in Computer Science. Vol. 206* (Berlin: Springer-Verlag)
- Stirling C 1986 A compositional reformulation of Owicki-Gries's partial correctness logic for a concurrent while language. In *Proceedings 13th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science. Vol. 226* (Berlin: Springer-Verlag)
- Stølen K 1990 *Development of parallel programs on shared data-structures*. Ph D thesis, Manchester University
- Woodcock J, Dickinson B 1988 Using VDM with rely and guarantee-conditions, experiences from a real project. In *Second VDM-Europe Symposium. Lecture Notes in Computer Science. Vol. 328* (Berlin: Springer-Verlag)
- Yi W 1990 Real-time behaviour of asynchronous agents. In *CONCUR'90. Lecture Notes in Computer Science. Vol. 458* (Berlin: Springer-Verlag)
- Zwarico A, Lee I 1985 Proving a network of real-time processes correct. In *Proceeding IEEE Real-Time Symposium*
- Zwiers J, de Roever W 1989 Predicates are predicate transformers: a unified compositional theory for concurrency. In *Proceeding 8th ACM Symposium on Principles of Distributed Computing* (New York: ACM Press)
- Zwiers J, de Roever W P, Van Emde Boas P 1984 Compositionality and concurrent networks: soundness and completeness of a proof system, Technical Report 57, University of Nijmegen
- Zwiers J 1989 Compositionality, concurrency and partial correctness. *Lecture Notes in Computer Science. Vol. 321* (Berlin: Springer-Verlag)