Design and verification in real-time distributed computing : an introduction to compositional methods

Citation for published version (APA):
Hooman, J. J. M., & Roever, de, W. P. (1990). Design and verification in real-time distributed computing : an introduction to compositional methods. In E. Brinksma, G. Scollo, & C. A. Vissers (Eds.), *Protocol Specification, Testing and Verification IX (Proceedings of the 9th IFIP WG 6.1 International Symposium, Enschede, The Netherlands, June 6-9, 1989)* (pp. 37-56). North-Holland Publishing Company.

Document status and date:
Published: 01/01/1990

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.

Download date: 04. Oct. 2023

Protocol Specification, Testing, and Verification, IX
E. Brinksma, G. Scollo and C.A. Vissers (Editors)
Elsevier Science Publishers B.V. (North-Holland)
© IFIP, 1990

Design and verification in real-time distributed computing: an introduction to compositional methods*

J.J.M. Hooman
W.P. de Roever

Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: mcvax!eutws1!wsinjh and mcvax!eutws1!wsinwpr

Abstract

Formal methods for the specification and verification of real-time systems can be considered from the viewpoint of expressibility (which properties can be specified), specification language (real-time temporal logic, first order assertions with time as explicit parameter), and programming features (time-out, communication mechanism, concurrency). We stress 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. This allows programming with such parts via their specifications rather than through their fully worked out code, thus separating the use of those parts as modules from their implementation. Therefore 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 code. We extend these compositional methods to real-time distributed computing. Compositional reasoning during top-down development is illustrated by an example concerning a watchdog timer.

1 Introduction

Numerous methods have been given for the specification and verification of (real-time) distributed programs. These methods can be considered from several viewpoints:

- The class of programs to which the method does apply: For instance, concurrent programs may communicate in many different ways; via shared variables or via message passing. We can distinguish between synchronous message passing—both sender and receiver wait until a partner is available—and asynchronous communication with many variations of buffering.
- The form of the correctness formulae used to specify and verify programs: Several methods use invariants, e.g. expressed in Temporal logic, or Hoare-triples (i.e. a program with pre- and post-condition) using first order logic.
- The properties which can be expressed: Real-time properties (expressing timing requirements, e.g. "communication every 5 time units"), safety properties (which can be falsified in finite time, e.g. "communication within 10 time units", or "nontermination"), or liveness properties (e.g. "eventually a communication will take place").

*This work was supported by ESPRIT Project 937: Debugging and Specification of Ada Real-Time Embedded Systems (DESCARTES).

- Is the method only applicable to complete program code or is it possible to verify design steps during the process of program development?

In this paper we concentrate on the last point, especially for real-time distributed message passing. We illustrate in section 3 the development from a-posteriori methods (which require the complete program text) towards compositional methods (supporting verify-while-design). Compositionality can be considered as a repercussion of hierarchical, structured, program derivation on specification and verification formalisms for (real-time) concurrent processes. A separation of concerns is proposed, implying a separation of the use of (and the reasoning about) a module from 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 construct 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 construct 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 as in [OG76,AFR80]). 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 [dR85]. 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, as in the present paper, 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.

When timing behaviour of a statement is considered, all factors concerning the execution of this statement which influence that timing behaviour must be expressible. E.g. in the present paper we make the maximal progress assumption with respect to distributed i/o-communication: no communication 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. In this real-time context, the introduction of priorities for processes on a single processor implies conceptually that certain statements which are ready to execute will not be executed on account of their lower 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. These aspects of reasoning formally about real-time and scheduling by means of priorities are addressed technically in section 3.4 and illustrated in section 4.

In section 2 we describe a real-time programming language with synchronous message passing and two types of parallel composition; one for parallel programs executing on a single processor and one for concurrent programs each executing on its own processor. In section 3 we give an example of a classical non-compositional method. We indicate how a compositional proof system (i.e. rules and axioms relating programs and specifications) can be achieved, and how such a method can be extended to reason about real-time properties. Compositional reasoning is illustrated, in section 4, by an example of a watchdog timer with several stages of development and using both notions of parallelism. Part of this example can also be found in [HW89] using real-time temporal logic. In section 5 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 A Real-Time Programming Language

We consider a real-time programming language which is akin to OCCAM [Occ88] and based on real-time variations of CSP [Hoa78] for which a formal denotational semantics has been given in [KSR⁺88]. A program in our language can have the following form:
$\ll S_{11} \,|||\, \cdots \,|||\, S_{1n_1} \gg \,\|\, \cdots \,\|\, \ll S_{m1} \,|||\, \cdots \,|||\, S_{mn_m} \gg$
where $\|$ expresses concurrent execution of the programs $\ll S_{i1} \,|||\, \cdots \,|||\, S_{in_i} \gg$, each on its own processor. The parallel operator $|||$ expresses that the programs involved are executed on a single processor. The brackets $\ll$ and $\gg$ indicate that no other processes are executed on a particular processor. Actions of parallel processes on a single processor are interleaved, on the basis of priorities assigned to statements. A statement **prio** p (S) assigns priority p to statement S. Statements without such an explicit priority have priority 0. A higher

number corresponds to a higher priority. There are two *primitive statements*:

- **skip** which does not take any execution time, and
- assignment $x := e$ which takes a certain execution time.

Processes communicate and synchronize by synchronous message passing via unidirectional channels which connect exactly two processes. Let D be a channel name, then our language contains the following *IO-statements*:

- **D!e** Output: send the value of expression e through channel D as soon as a corresponding input command is available.
- **D?x** Input: receive a value via channel D and assign this value to the variable x. Similar to the output command, this statement has to wait for a corresponding partner before a (synchronous) communication can take place.

In general, a processor can be either at a *scheduling* point or at a *nonscheduling* point. The priority of a statement is only considered at scheduling points; then a statement can only execute if there are no other statements with a higher priority on the same processor which request processor time. In case of equal priorities a nondeterministic choice is made (to abstract from specific scheduling policies). The execution of a statement at a *nonscheduling point* can not be interrupted, not even by requesting statements with a higher priority. A statement requests processor time until it has the highest priority; then a primitive statement can start executing at a scheduling point. An IO-statement with highest priority either waits for a partner (allowing lower priorities to execute) or starts the communication if a partner is available. In this paper only the start and termination points of primitive and IO-statements are considered to be scheduling points.

Furthermore, our syntax contains the following *composite commands*:

- Sequential composition $S_1; S_2$.
- A guarded command $[b_1; IO_1 \rightarrow S_1 [] \cdots [] b_n; IO_n \rightarrow S_n [] b; \mathbf{delay}\, d \rightarrow S]$, with IO_i denoting an IO-statement. It is executed as follows: first wait until an IO-command of the open guards (i.e. for which the boolean b_i evaluates to true) can be executed and then continue with the corresponding S_i. If the delay guard is open (b evaluates to true) and no IO-guard can be taken within d time units, then S is executed. This makes it possible to model a *time-out*, i.e. to restrict the waiting period for certain communications.

  Example 2.1 Consider $[D?x \rightarrow S_1 [] \mathbf{delay}\, 5 \rightarrow S_2]$; if the D-communication can not be executed within 5 time units then the delay-branch is taken and S_2 requests execution time. Note that in guards with a boolean expression equivalent to *true* this boolean is often omitted. ∎

- An iteration statement **$*G$** repeats the 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.

2.1 Basic Timing Assumptions

In order to determine the real-time behaviour of programs, we have to make assumptions about the execution time of atomic statements and the overhead associated with composite constructs. Moreover, bounds must be given on how long a process is allowed to wait with the execution of a primitive statement when a processor is available, and with the execution of an IO-statement when processor and communication partner are available. In this work we assume *maximal progress* which means that a process never waits unnecessarily; if execution can proceed it will do so immediately. Note that there are two reasons for a process to wait:

- Wait to execute an IO-statement because no communication partner is available. By the maximal progress assumption, two statements $D!e$ and $D?x$ are not allowed to wait simultaneously if both can execute on their processor.
- Wait to execute an atomic statement because the processor is not available. The maximal progress constraint implies that if a processor is idle, then no statement on that processor requests execution time.

Throughout this paper we use $\equiv$ to express syntactic equality.

3 Compositionality and Real-Time

In section 3.1 we explain the principles of traditional non-compositional methods. The development towards compositional proof systems based on Hoare triples is described in section 3.2. Extensions of these formulae with invariants are discussed in section 3.3. Since none of the methods described in the sections 3.1— 3.3 can express timing behaviour of programs, we do not use the operator $|||$ nor the brackets $\ll$, $\gg$ in these sections. In section 3.4 the compositional method based on explicit assumption-commitment reasoning is adapted to real-time.

3.1 Non-Compositional Methods

Classical verification methods for parallel processes, such as [OG76] for shared variable communication and [AFR80,LG81] for synchronous message passing, consist of two stages. First a *local* correctness proof is given for each of the sequential processes 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.

As an example, we consider in more detail the method of Apt, Francez and de Roever [AFR80] for synchronous message passing. This method is based on *Hoare triples*, that are correctness formulae of the form $\{p\}\ S\ \{q\}$ with the following meaning: if we start program S in a state satisfying assertion p (the pre-condition) and if S terminates then assertion q (the post-condition) holds for the termination state. E.g. $\{x = 5\}\ x := x + 1\ \{x = 6\}$.

First we indicate how a proof system can be formulated to derive such Hoare triples for sequential programs. With $q[e/x]$ denoting textual substitution of e for each free occurrence of x in assertion q, 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 3.1 (sequential composition) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1; S_2\ \{q\}}$

By such a rule the formula below the line can be derived from the formulae above the line. Soundness of the rule is proven 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 3.2 (consequence) $\dfrac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$

To illustrate the rule for parallel composition in [AFR80], we consider the proof of $\{y = 3\}\ B?x\ ;\ x := x + 1\ ;\ E!(x + 2) \parallel B!y\ ;\ E?y\ ;\ y := y + 2\ \{x = 4 \wedge y = 8\}$.

In the first stage we attach assertions to all locations in the program text of the two processes, leading to so called *proof outlines*:
$\{true\}\ B?x\ \{x = 3\}\ ;\ x := x + 1\ \{x = 4\}\ ;\ E!(x + 2)\ \{x = 4\}$, and
$\{y = 3\}\ B!y\ \{y = 3\}\ ;\ E?y\ \{y = 6\}\ ;\ y := y + 2\ \{y = 8\}$.
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 $B?x$ and $E?y$ 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 $\{p_1\}\ D?x\ \{q_1\}$ and $\{p_2\}\ D!e\ \{q_2\}$ in the proof outlines of two processes we have to prove $\{p_1 \wedge p_2\}\ D?x \parallel D!e\ \{q_1 \wedge q_2\}$, which is equivalent to $\{p_1 \wedge p_2\}\ x := e\ \{q_1 \wedge q_2\}$. In our example this leads to the proof obligations:
$\{true \wedge y = 3\}\ B?x \parallel B!y\ \{x = 3 \wedge y = 3\}$ and
$\{y = 3 \wedge x = 4\}\ E?y \parallel E!(x + 2)\ \{y = 6 \wedge x = 4\}$ 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

[1]In the full method of [AFR80] auxiliary variables and a global invariant are used to make the method complete, i.e. to guarantee that any valid Hoare triple can be proven.

pre-condition $true \wedge y = 3$ and 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 [AFR80] is removed by disallowing implicit assumptions in the post-conditions of IO-statements. The local proof of a sequential program should be valid in any arbitrary environment. To derive a required (valid) post-condition for the complete program, we use a *history variable* h which denotes the communication history of the complete program. A (communication) *history* is a sequence of records (D, v) where D is a channel name and v a value. E.g. $<(D,5),(E,6),(B,8),(E,0)>$ is a history expressing four communications: first one via channel D with value 5, then a communication via E with value 6, etc. Let $<>$ denote the empty sequence. History variable h does not appear in the program, but it is updated implicitly in the semantics of IO-statements. This leads to the following formulae:

- For an output command e.g. $\{h =< (D,5) >\}\ E!6\ \{h =< (D,5),(E,6) >\}$.
- For an input statement $D?x$ we can only express in the post-condition that there exists a value which is communicated via D and assigned to x. For instance, $\{h =<>\}\ D?x\ \{\exists v : h =< (D,v) > \wedge x = v\}$.

Example 3.2 Consider the parallel composition of $D!5$ and $D?x$, using Hoare triples $\{h =<>\}\ D!5\ \{h =< (D,5) >\}$ and $\{h =<>\}\ D?x\ \{\exists v : h =< (D,v) > \wedge x = v\}$.
Suppose the pre- and post-condition after parallel composition are obtained by taking the conjunction of, resp., pre- and post-conditions of the sequential programs. Then
$\{h =<>\}\ D!5 \parallel D?x\ \{h =< (D,5) > \wedge \exists v : h =< (D,v) > \wedge x = v\}$.
Since the post-condition implies $v = 5 \wedge x = v$, and hence $x = 5$, the consequence rule leads to $\{h =<>\}\ D!5 \parallel D?x\ \ \{x = 5\}$. By a so called *substitution rule* (not given in this paper), we could substitute $<>$ for h in the pre-condition, thus obtaining pre-condition *true*. ▮

Example 3.2 suggests the following rule: $\dfrac{\{p_1\}\ S_1\ \{q_1\},\ \{p_2\}\ S_2\ \{q_2\}}{\{p_1 \wedge p_2\}\ S_1 \| S_2\ \{q_1 \wedge q_2\}}$

Example 3.3 Consider again $S_1 \equiv B?x;\ x := x + 1;\ E!(x + 2) \parallel S_2 \equiv B!y;\ E?y;\ y := y + 2$.
First derive the following Hoare triples: $\{h =<>\}\ B?x\ \{\exists v_1 : h =< (B, v_1) > \wedge x = v_1\}$,
$\{\exists v_1 : h =< (B, v_1) > \wedge x = v_1\}\ x := x + 1\ \{\exists v_1 : h =< (B, v_1) > \wedge x = v_1 + 1\}$, and
$\{\exists v_1 : h =< (B, v_1) > \wedge x = v_1 + 1\}\ E!(x+2)\ \{\exists v_1 : h =< (B, v_1),(E, v_1 + 3) > \wedge x = v_1 + 1\}$.
By two applications of the sequential composition rule we obtain:
$\{h =<>\}\ B?x\ ;\ x := x + 1\ ;\ E!(x + 2)\ \{\exists v_1 : h =< (B, v_1),(E, v_1 + 3) > \wedge x = v_1 + 1\}$.
Similarly, $\{h =<> \wedge y = 3\}\ B!y;\ E?y;\ y := y + 2\ \{\exists v_2 : h =< (B,3),(E, v_2) > \wedge y = v_2 + 2\}$.
Then the parallel composition rule above leads to

$$\{h =<> \wedge y = 3\}\ S_1 \| S_2\ \{\ \exists v_1 : h =< (B, v_1),(E, v_1 + 3) > \wedge x = v_1 + 1 \wedge$$
$$\exists v_2 : h =< (B,3),(E, v_2) > \wedge y = v_2 + 2\}.$$

The post-condition implies $v_1 = 3 \wedge v_2 = v_1 + 3 \wedge x = v_1 + 1 \wedge y = v_2 + 2$, which leads to

$x = 4 \wedge y = 8$. Thus, by the consequence rule, $\{h =<> \wedge y = 3\}\ S_1\|S_2\ \{x = 4 \wedge y = 8\}$. (Again $h =<>$ in the pre-condition can be removed by the substitution rule.) ∎

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 $S_1 \equiv B!0\ ;\ E?x \parallel S_2 \equiv B?y\ ;\ D!(y+1) \parallel S_3 \equiv D?z\ ;\ E!(z+1)$. Following the previous example, we could first prove:
$\{h =<>\}\ S_1\ \{q_1 \equiv \exists v_1 : h =< (B,0),(E,v_1) > \wedge x = v_1\}$, and
$\{h =<>\}\ S_2\ \{q_2 \equiv \exists v_2 : h =< (B,v_2),(D,v_2{+}1) > \wedge y = v_2\}$. But then the conjunction of q_1 and q_2 implies *false* whereas $S_1\|S_2\|S_3$ 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. Zwiers [Zwi88], however, shows that concise and simple rules 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 channel name not in *cset*. E.g. if $h =< (B,0),(D,1),(E,3) >$ then $h_{\{D\}} =< (D,1) >$, $h_{\{D,E\}} =< (D,1),(E,3) >$, and $h_{\{F\}} =<>$. Henceforth we write h_D, h_{DE}, and h_F instead of, resp., $h_{\{D\}}$, $h_{\{D,E\}}$, and $h_{\{F\}}$.

At parallel composition of S_1 and S_2 we require that the post-condition of S_i only refers to h via projections on the channels occurring in S_i, for $i = 1,2$. If, moreover, the post-condition of S_1 (resp. S_2) does not refer to program variables of S_2 (resp. S_1), then the following rule for parallel composition is sound:

Rule 3.3 (parallel composition) $\dfrac{\{p_1\}\ S_1\ \{q_1\},\ \{p_2\}\ S_2\ \{q_2\}}{\{p_1 \wedge p_2\}\ S_1\|S_2\ \{q_1 \wedge q_2\}}$

Observe that this is a compositional rule because a Hoare triple for $S_1\|S_2$ can be derived without knowing the internal structure of S_1 and S_2. Recall that Hoare triples are not sufficient to formulate the cooperation test but that proof outlines are required. In the rule above this test is replaced by a simple syntactic check on the Hoare triples for the parallel components.

Except for bottom-up verification such a rule can be used for top-down development. Therefore, consider $\{p\}\ S\ \{q\}$ as a specification for a program S. Suppose we decide to implement S as $S_1\|S_2$. Then S_1 and S_2 can be implemented independently, using specifications $\{p_1\}\ S_1\ \{q_1\}$ and $\{p_2\}\ S_2\ \{q_2\}$ provided $p \rightarrow p_1 \wedge p_2$, $q_1 \wedge q_2 \rightarrow q$ and certain requirements on the post-conditions hold.

3.3 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 state. 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 can not be used to express the communication interface. 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 $C : \{p\}\ S\ \{q\}$ 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.

Yet the framework explained so far is not satisfactory. In the form of the specification of a process there is no separate component describing the behaviour of its environment, whereas in general the behaviour of a process depends heavily on its environment. Especially when specifying so called reactive processes [HP85], which have an intensive relation with their environment, we want to specify a process relative to explicit assumptions about its environment. 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 $(A, C) : \{p\}\ S\ \{q\}$, 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.

In the following examples, $seq_1 \preceq seq_2$ expresses that sequence seq_1 is an initial prefix of sequence seq_2.

Example 3.5 Now assumptions about the values sent by the environment can be expressed explicitly. For instance, $h_D \preceq< (D,3) >$, which expresses that the history of channel D is a prefix of $< (D,3) >$, can be used as an assumption as follows:
$(h_D \preceq< (D,3) >, true) : \{true\}\ D?x\ \{x = 3\}$.
This assumption can be used for a commitment about the next communication:
$(h_D \preceq< (D,3) >, h_B \preceq< (B,4) >) : \{true\}\ D?x\ ;\ B!(x+1)\ \{x = 3\}$. ∎

A proof system for these assumption-commitment based formulae has been given in [Hoo89]. In this paper we discuss mainly the proof obligations for assumptions and commitments at parallel composition. Consider the parallel composition $S_1\|S_2$, and assume 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 $S_1\|S_2$? 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 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 for A_1: $A \wedge C_2 \rightarrow A_1$.

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 $A \equiv \mathit{true}$ lead to circular reasoning, e.g. $A_1 \rightarrow C_1 \rightarrow A_2 \rightarrow C_2 \rightarrow A_1$. 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 preceeding communications. This inductive step inside the meaning of formulas is sufficient to avoid circularity (see [MC81]).

As in Rule 3.3 we can take the conjunction of pre-conditions and post-conditions and also of commitments, provided 1) the assertions A_i, C_i, p_i and q_i of S_i refer only to h via projections on the channels, and 2) p_i and q_i do not refer to program variables of the other process. (Program variables are not allowed in A_i and C_i.) With these constraints, the following rule for parallel composition is valid:

Rule 3.4 (par. comp. A-C)
$$\frac{(A_1, C_1) : \{p_1\}\ S_1\ \{q_1\},\ (A_2, C_2) : \{p_2\}\ S_2\ \{q_2\} \qquad A \wedge C_1 \rightarrow A_2,\ A \wedge C_2 \rightarrow A_1}{(A, C_1 \wedge C_2) : \{p_1 \wedge p_2\}\ S_1 \| S_2\ \{q_1 \wedge q_2\}}$$

Example 3.6 Consider $S_1 \equiv B?x\,;\ x := x+1\,;\ E!(x+2) \parallel S_2 \equiv D?y\,;\ B!y\,;\ E?y\,;\ y := y+2$.
Then for S_1 and S_2 we can derive
$(A_1 \equiv h_B \preceq <(B,3)>\,,\ C_1 \equiv h_E \preceq <(E,6)>) : \{h_{BE} = <>\}\ S_1\ \{x = 4\}$, and
$(A_2 \equiv h_D \preceq <(D,3)> \wedge h_E \preceq <(E,6)>, C_2 \equiv h_B \preceq <(B,3)>) : \{h_{BDE} = <>\}\ S_2\ \{y = 8\}$.
Since S_1 and S_2 communicate with each other via the channels B and E, we take for $S_1 \| S_2$ the assumption about the remaining channel: $A \equiv h_D \preceq <(D,3)>$.
Then $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$, thus the parallel composition rule leads to
$(A, C_1 \wedge C_2) : \{h_{BDE} = <>\}\ S_1 \| S_2\ \{x = 4 \wedge y = 8\}$. ▮

3.4 Extension to Real-Time

In this section we discuss the extension of compositional methods, in particular the assumption/commitment formalism described in the previous section, to real-time specifications containing timing requirements.

We express the timing behaviour of a program from the viewpoint of an external observer with his own clock. Thus at the level of reasoning there is a conceptual global clock. Since our global clock is not incorporated in the distributed system itself, it does not impose any synchronization upon processes. In the specification a special variable *time* is introduced which refers to this external clock. E.g. if an assignment takes between 3 and 5 time units, then $\{x = 5 \wedge \mathit{time} = 2\}\ x := x + 1\ \{x = 6 \wedge 5 < \mathit{time} < 7\}$. In this paper we assume that *time* ranges over a dense time domain which includes the natural numbers with the standard ordering $<$, e.g. the rational numbers.

To express the communication behaviour of a program, our assertion language contains primitives to express when a communication via a certain channel takes place, and which

value is transmitted. Recall from section 2 that the maximal progress assumption imposes certain constraints on waiting before a communication takes place. As already explained in section 1, this implies that we must be able to express when a process is waiting to communicate via a particular channel. In this paper we use the following primitives:

- *comm via D at t with value v*, to denote that a communication via channel D with value v takes place at t.
- *wait to D! at t*, to denote that a process is waiting to send a message via D at t.
- *wait to D? at t*, to denote that a process is waiting to receive via D at t.

Note that, in general, communication takes a certain period of time, and hence there will be an interval of points at which a process is communicating via a particular channel with a certain value. With these primitives the *maximal progress* constraint can be expressed as follows, for every channel D;

Axiom 3.2 (MP) $\forall t : \neg(\mathit{wait\ to\ D?\ at\ t}\ \wedge \mathit{wait\ to\ D!\ at\ t}\ )$

Furthermore we will later use the following axiom which excludes that a program is simultaneously waiting to communicate and communicating on the same channel:

Axiom 3.3 (Excl) $\forall t : \neg[(\mathit{wait\ to\ D?\ at\ t}\ \vee \mathit{wait\ to\ D!\ at\ t}\ ) \wedge \mathit{comm\ via\ D\ at\ t}]$

The following abbreviations are frequently used:

- $\mathit{comm\ via\ D\ at\ t} \equiv \exists v : \mathit{comm\ via\ D\ at\ t\ with\ value\ v}$
- $\mathit{(wait\ to)\ comm\ via\ D!\ at\ t} \equiv \mathit{comm\ via\ D\ at\ t} \vee \mathit{wait\ to\ D!\ at\ t}$
- $\mathit{wait\ to\ D!\ during}\ (t_0, t_2) \equiv \forall t_1, t_0 < t_1 < t_2 : \mathit{wait\ to\ D!\ at}\ t_1$
- $\mathit{wait\ to\ D!\ at\ t\ Until\ comm\ via\ D} \equiv$
  $\forall t_1 \geq t : \mathit{wait\ to\ D!\ at}\ t_1 \vee\ \exists t_2\ \forall t_1, t \leq t_1 < t_2 : \mathit{wait\ to\ D!\ at}\ t_1 \wedge \mathit{comm\ via\ D\ at}\ t_2$

From these definitions we derive the following lemma.

Lemma 3.1 *wait to D! at t Until comm via D* $\rightarrow$ *(wait to) comm via D! at t*

Example 3.7 Consider processes P_1 and P_2 with specifications
$\{p_1\}\ P_1\ \{$*wait to D? at 5 Until comm via D*$\}$ and
$\{p_2\}\ P_2\ \{$*wait to D! at 5 Until comm via D*$\}$.
Then the rule for parallel composition leads to the following post condition for $P_1 \parallel P_2$:
wait to D? from 5 Until comm via D $\wedge$ *wait to D! from 5 Until comm via D*. Now the maximal progress requirement implies that there is no point of time with waiting for input and waiting for output via channel D. Hence the D communication must start at time 5. ▮

Considering uniprocessor implementations with scheduling based on priorities, again our assertion language must be extended to achieve compositionality. Recall from section 2 that in this case we have to express when a process is executing and when it requests processor time with a certain priority. Therefore we introduce the following predicates:

- $\mathit{exec}(t, \infty)$ which is true when the process is executing at a non-scheduling point t.
- $\mathit{exec}(t, p)$, for $p \neq \infty$, which is true when the process is executing at a scheduling point t with priority p.
- $\mathit{req}(t, p)$ which is true when the process requests processor time at t with priority p.

Furthermore, a priority is associated with waiting for a communication:

- *(wait to) comm via D! at t with prio p*

The following general axiom expresses that if a statement is executing with a certain priority, then no IO-statement is waiting to communicate or communicating with a lower priority:

Axiom 3.4 (Prio) $exec(t,p_1) \wedge p_1 > p_2 \rightarrow \neg$*(wait to) comm via D! at t with prio* p_2

In the assumption/commitment formalism, we can now express 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, and derive the termination time in the post-condition (assume the communication takes 1 time unit):

$$\begin{aligned} (A &\equiv \textit{(wait to) comm via } D! \textit{ at } 5 \wedge \forall t, 3 \leq t < 5 : \neg \textit{comm via } D \textit{ at } t, \\ C &\equiv \forall t, 3 \leq t < 5 : \textit{wait to } D? \textit{ at } t \wedge \forall t, 5 < t < 6 : \textit{comm via } D \textit{ at } t) : \\ &\quad \{time = 3\}\ D?x\ \{time = 6\} \end{aligned}$$

Note that, due to the maximal progress constraint, a communication takes place as soon as both process and environment are ready for input and, resp., output.

When adding timing primitives to the assertion language, Rule 3.4 for the parallel composition $S_1 \| S_2$ has to be adapted slightly. Therefore observe that in the post-conditions q_1 and q_2 of, resp., S_1 and S_2 the special variable *time* denotes the termination time of, resp., S_1 and S_2. Since these termination times will be different in general (and then $q_1 \wedge q_2$ could implies *false*), we substitute logical variables t_1 (resp. t_2) for *time* in assertion q_1 (resp. q_2) before the conjunction is taken. Hence Rule 3.4 is adapted as follows: replace $q_1 \wedge q_2$ in the post-condition of $S_1 \| S_2$ by q, and add to the conditions above the line $q_1[t_1/time] \wedge q_2[t_2/time] \wedge time = max(t_1, t_2) \rightarrow q$ because the parallel construct terminates if both processes have terminated.

Example 3.8 Consider the following specifications (**action** d is used to represent an internal actions which takes d time units and assume communications take one time unit):

$$\begin{aligned} (A_1 &\equiv \textit{(wait to) comm via } B? \textit{ at } 2 \wedge \forall t, 0 \leq t < 2 : \neg \textit{comm via } B \textit{ at } t \wedge \\ &\quad \textit{(wait to) comm via } D? \textit{ at } 6 \wedge \forall t, 3 \leq t < 6 : \neg \textit{comm via } D \textit{ at } t, \\ C_1 &\equiv \forall t, 2 < t < 3 : \textit{comm via } B \textit{ at } t \wedge \textit{wait to } D! \textit{ at } 3 \textit{ Until comm via } D \wedge \\ &\quad [\forall t : (\exists v : \textit{comm via } D \textit{ at } t \textit{ with value } v) \rightarrow v = 5]) : \\ &\quad \{time = 0\}\ S_1 \equiv \ll B!1 \,;\, D!5 \,;\, \textbf{action } 2 \gg\ \{time = 9\}, \text{ and} \\ (A_2 &\equiv \textit{wait to } D! \textit{ at } 3 \textit{ Until comm via } D \wedge \\ &\quad [\forall t : (\exists v : \textit{comm via } D \textit{ at } t \textit{ with value } v) \rightarrow v = 5], \\ C_2 &\equiv \textit{(wait to) comm via } D? \textit{ at } 6 \wedge \forall t, 3 \leq t < 6 : \neg \textit{comm via } D \textit{ at } t \wedge \\ &\quad \forall t, 6 < t < 7 : \textit{comm via } D \textit{ at } t) : \\ &\quad \{time = 0\}\ S_2 \equiv \ll \textbf{action } 6 \,;\, D?x \gg\ \{time = 7 \wedge x = 5\} \end{aligned}$$

Take for $S_1 \| S_2$ the following assumption: $A \equiv$ *(wait to) comm via B? at* $2 \wedge \forall t, 0 \leq t < 2 : \neg$*comm via B at t*, then clearly: $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$. The parallel composition rule leads to $(A, C_1 \wedge C_2) : \{time = 0\}\ S_1 \| S_2\ \{time = 9 \wedge x = 5\}$. ∎

For the uniprocessor parallel composition $|||$ we follow the same scheme, with some changes to deal with the *exec* and *req* primitives. Observe that the commitments C_1 and C_2 in general express different properties of these predicates. Therefore rename *exec* in C_1 and C_2 by, resp., predicates e_1 and e_2. Since $S_1 ||| S_2$ executes iff S_1 or S_2 executes, we add the property $exec \leftrightarrow e_1 \vee e_2$, which is an abbreviation of $\forall p\, \forall t : exec(t,p) \leftrightarrow e_1(t,p) \vee e_2(t,p)$. Similar changes are made for the *req* predicate. This leads to the following conditions for deriving a pair (A, C) for $S_1 ||| S_2$ from pairs (A_1, C_1) for S_1 and (A_2, C_2) for S_2:

- $A[e_1/exec, r_1/req] \wedge C_1[e_2/exec, r_2/req] \wedge (e_1 \vee e_2 \leftrightarrow exec) \wedge (r_1 \vee r_2 \leftrightarrow req) \rightarrow A_2$, and
  $A[e_1/exec, r_1/req] \wedge C_2[e_2/exec, r_2/req] \wedge (e_1 \vee e_2 \leftrightarrow exec) \wedge (r_1 \vee r_2 \leftrightarrow req) \rightarrow A_1$

- $C_1[e_1/exec, r_1/req] \wedge C_2[e_2/exec, r_2/req] \wedge (e_1 \vee e_2 \leftrightarrow exec) \wedge (r_1 \vee r_2 \leftrightarrow req) \rightarrow C$

4 Example Watchdog Timer

To illustrate our formalism with an example, consider the network pictured in Fig. 1. Process

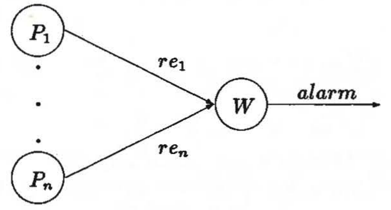

Figure 1: Watchdog Timer Network

W is a "watchdog" process: its job is to ensure that processes $P_1, \ldots, P_n$ are functioning properly. Each P_i indicates that it is functioning by sending a reset-signal on channel re_i at least every v_i (say) time units. Process W is always ready to receive signals on any re_i. If there is no communication on a particular re_k for v_k or more time units, then, within K (say) time units, W sends a warning on channel *alarm*. In section 4.1 we give a specification for process W. Given specifications for the P_i, we prove for $P_1 \| \cdots \| P_n \| W$ that one of the P_i is not functioning correctly iff W tries to send on channel *alarm*. This is verified using our proof system without knowing the implementations of $P_1, \ldots, P_n$ and W. To demonstrate program design from a specification, in section 4.2 W is implemented as a parallel composition $W_1 \| \cdots \| W_n \| S$ and we show which specifications for W_i and S are sufficient to derive the specification of W. Next W_i and S are, independently, implemented, satisfying the corresponding implementations. In section 4.3, we show that P_i can be implemented as $\ll$ **prio** 2 (B_i) $|||$ **prio** 1 $(WH_i) \gg$ where B_i performs the real task of P_i and WH_i is a watchdog handler which sends signal via re_i as long as B_i is functioning properly.

In this example we concentrate on the assumption-commitment reasoning. Henceforth all specifications have pre-condition $time = 0$ and post-condition *true*.

4.1 Specification of the Watchdog Timer

In this section we give a formal specification for the watchdog timer W which tries to communicate via channel *alarm* 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 for input via one of the re_i. Furthermore, if there is a waiting period for input via re_k of v_k time units, then, for some constant K, W starts waiting to output on channel *alarm* until the actual communication takes place. Hence we specify W as follows.

$$\begin{aligned}
(A^W &\equiv \mathit{true}, \\
C_0^W &\equiv [\mathit{wait\ to\ re_k?\ during}\ (t_2 - v_k, t_2) \rightarrow \\
&\quad \exists t_0 \leq t_2 + K : \mathit{wait\ to\ alarm!\ at}\ t_0\ \mathit{Until\ comm\ via\ alarm}] \wedge \\
C_1^W &\equiv [\mathit{(wait\ to)\ comm\ via\ alarm!\ at}\ t_1 \rightarrow \\
&\quad \exists k\, \exists t_0 \leq t_1 : \mathit{wait\ to\ re_k?\ during}\ (t_0 - v_k, t_0)]) :\ W
\end{aligned}$$

By convention, free variables in such a specification (e.g. t_1 and t_2) are universally quantified.

We prove that W tries to send a message via *alarm* iff there is an error in one of the processes P_i. Therefore we use the following specification for the P_i, using a predicate $error_i$ which represents some erroneous behaviour of P_i. For all i:

$$\begin{aligned}
(A^{P_i} &\equiv \mathit{true}, \\
C^{P_i} &\equiv [\mathit{error}_i \leftrightarrow \exists t\, \forall t_1, t - v_i < t_1 < t : \neg\mathit{(wait\ to)\ comm\ via\ re_i!\ at}\ t_1\ ]) :\ P_i
\end{aligned}$$

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 neither communicating via re_i nor waiting to communicate via re_i. Given our specifications for $P_1, \ldots, P_n$ and W, we try to prove that $P_1 \| \cdots \| P_n \| W$ has commitment $\exists k : error_k \leftrightarrow \exists t :$ *(wait to) comm via alarm! at* t. We can apply parallel composition n times and obtain $(\mathit{true}, C_1^W \wedge C_2^W \wedge \bigwedge_{i=1}^n C^{P_i}) : \ldots P_1 \| \cdots \| P_n \| W \ldots$
First prove $\exists k : error_k \leftarrow \exists t :$ *(wait to) comm via alarm! at* t.

$$\begin{array}{lll}
 & & \exists t : \mathit{(wait\ to)\ comm\ via\ alarm!\ at}\ t \\
\rightarrow & \{C_1^W\} & \exists t\, \exists k\, \exists t_0 \leq t : \mathit{wait\ to\ re_k?\ during}\ (t_0 - v_k, t_0) \\
\rightarrow & \{\text{definition}\} & \exists k\, \exists t_0\, \forall t_1, t_0 - v_k < t_1 < t_0 : \mathit{wait\ to\ re_k?\ at}\ t_1 \\
\rightarrow & \{\text{MP, Excl}\} & \exists k\, \exists t_0\, \forall t_1, t_0 - v_k < t_1 < t_0 : \neg\mathit{wait\ to\ re_k!\ at}\ t_1 \wedge \neg\mathit{comm\ via\ re_k\ at}\ t_1 \\
\rightarrow & \{\text{definition}\} & \exists k\, \exists t_0\, \forall t_1, t_0 - v_k < t_1 < t_0 : \neg\mathit{(wait\ to)\ comm\ via\ re_k!\ at}\ t_1 \\
\rightarrow & \{C^{P_k}\} & \exists k : \mathit{error}_k
\end{array}$$

Next we try to prove $\exists k : error_k \rightarrow \exists t :$ *(wait to) comm via alarm! at* t.
From $\exists k : error_k$ we obtain, by C^{P_k}, $\exists k \exists t \forall t_1, t - v_k < t_1 < t : \neg$*(wait to) comm via* re_k*! at* t_1.
Thus $\exists k\, \exists t\, \forall t_1, t - v_k < t_1 < t : \neg$*comm via* re_k *at* t_1, but with the current specification of W 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:

$$\begin{aligned}
(A^W &\equiv \mathit{true}, \\
C_1^W &\equiv [\mathit{(wait\ to)\ comm\ via\ alarm!\ at}\ t_1 \rightarrow \\
&\quad \exists k\, \exists t_0 \leq t_1 : \mathit{wait\ to\ re_k?\ during}\ (t_0 - v_k, t_0)] \wedge \\
C_2^W &\equiv [\exists t\, \forall t_1, t - v_k < t_1 < t : \neg\mathit{comm\ via\ re_k\ at}\ t_1 \rightarrow \\
&\quad \exists t_0 \leq t + K : \mathit{wait\ to\ alarm!\ at}\ t_0\ \mathit{Until\ comm\ via\ alarm}]) :\ W
\end{aligned}$$

Note that C_0^W follows from C_2^W. Now the proof proceeds as follows, for all k,

$$\begin{array}{lll}
 & & \mathit{error}_k \\
\rightarrow & \{C^{P_k}\} & \exists t\, \forall t_1, t - v_k < t_1 < t : \neg\mathit{(wait\ to)\ comm\ via\ re_k!\ at}\ t_1 \\
\rightarrow & \{\text{definition}\} & \exists t\, \forall t_1, t - v_k < t_1 < t : \neg\mathit{comm\ via\ re_k\ at}\ t_1 \\
\rightarrow & \{C_2^W\} & \exists t_0 \leq t + K : \mathit{wait\ to\ alarm!\ at}\ t_0\ \mathit{Until\ comm\ via\ alarm} \\
\rightarrow & \{\text{Lemma 3.1}\} & \exists t_0 : \mathit{(wait\ to)\ comm\ via\ alarm!\ at}\ t_0
\end{array}$$

This reasoning allows us to verify properties of $P_1 \| P_2 \| \cdots \| P_n \| W$ (and to discover incompleteness of the specification) using the specifications for the components, without knowing the implementations of these processes.

4.2 Implementing the Watchdog Timer

Next we implement process W as a parallel composition, $W \equiv W_1 \| \cdots \| W_n \| S$, where each W_i is a watchdog timer for process P_i. W_i signals process S via channel al_i as soon as there is no communication on re_i for at least v_i time units. Process S waits for a signal on any of the al_i's; after receipt of a signal it tries to send a message on *alarm* (see Fig. 2).

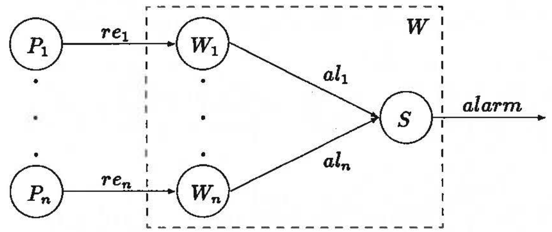

Figure 2: Implementation Watchdog Timer

We prove that the following specifications for W_i and S are sufficient to derive the specification for W. The specification for W_i expresses that W_i tries to communicate via al_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 al_i within a certain time bound K_i.

$$\begin{aligned}
(A^{W_i} &\equiv \mathit{true}, \\
C_1^{W_i} &\equiv [\mathit{(wait\ to)\ comm\ via\ al_i!\ at}\ t_1 \rightarrow \\
&\quad \exists t_0 \leq t_1 : \mathit{wait\ to\ re_i?\ during}\ (t_0 - v_i, t_0)] \wedge \\
C_2^{W_i} &\equiv [\exists t\, \forall t_1, t - v_k < t_1 < t : \neg\mathit{comm\ via\ re_i\ at}\ t_1 \rightarrow \\
&\quad \exists t_0 \leq t + K_i : \mathit{(wait\ to)\ comm\ via\ al_i!\ at}\ t_0]) :\ W_i
\end{aligned}$$

Let K_S be a constant (e.g. denoting the maximum duration of a communication). The specification for S asserts that it tries to send a message via *alarm* only if there was a preceding communication via one of the al_k. If S does not try to communicate nor communicates via one of the al_k at a certain point of time, then within K_S time units it will wait to communicate via alarm until the actual communication can be performed.

$$\begin{array}{lcl}
(A^S & \equiv & \mathit{true}, \\
C_1^S & \equiv & [\mathit{(wait\ to)\ comm\ via\ alarm!\ at}\ t_1 \rightarrow \exists k\, \exists t_0 \leq t_1 : \mathit{comm\ via}\ al_k\ \mathit{at}\ t_0] \wedge \\
C_2^S & \equiv & [\neg\ \mathit{wait\ to}\ al_k?\ \mathit{at}\ t_2 \rightarrow \\
 & & \exists t_0 \leq t_2 + K_S : \mathit{wait\ to\ alarm!\ at}\ t_0\ \mathit{Until\ comm\ via\ alarm}]) :\ S
\end{array}$$

We show that $W_1 \| \cdots \| W_n \| S$ satisfies the specification of W. We can take, by repeated application of the parallel composition rule, the conjunction of commitments: $\bigwedge_{i=1}^{n}(C_1^{W_i} \wedge C_2^{W_i}) \wedge C_1^S \wedge \wedge C_2^S$. This implies C_1^W as follows:

$$\begin{array}{lcl}
 & & \mathit{(wait\ to)\ comm\ via\ alarm!\ at}\ t_1 \\
\rightarrow & \{C_1^S\} & \exists k\, \exists t_3 \leq t_1 : \mathit{comm\ via}\ al_k\ \mathit{at}\ t_3 \\
\rightarrow & \{\text{definition}\} & \exists k\, \exists t_3 \leq t_1 : \mathit{(wait\ to)\ comm\ via}\ al_k!\ \mathit{at}\ t_3 \\
\rightarrow & \{C_1^{W_k}\} & \exists k\, \exists t_3 \leq t_1 \exists t_0 \leq t_3 : \mathit{wait\ to}\ re_k?\ \mathit{during}\ (t_0 - v_k, t_0) \\
\rightarrow & \{t_0 \leq t_3 \leq t_1\} & \exists k\, \exists t_0 \leq t_1 : \mathit{wait\ to}\ re_k?\ \mathit{during}\ (t_0 - v_k, t_0)
\end{array}$$

Next we prove C_2^W.

$$\begin{array}{lcl}
 & & \exists t\, \forall t_1, t - v_k < t_1 < t : \neg \mathit{comm\ via}\ re_k\ \mathit{at}\ t_1 \\
\rightarrow & \{C_2^{W_k}\} & \exists t_2 \leq t + K_k : \mathit{(wait\ to)\ comm\ via}\ al_k!\ \mathit{at}\ t_2 \\
\rightarrow & \{\text{definition}\} & \exists t_2 \leq t + K_k : \mathit{wait\ to}\ al_k!\ \mathit{at}\ t_2 \vee \mathit{comm\ via}\ al_k!\ \mathit{at}\ t_2 \\
\rightarrow & \{\text{MP, Excl}\} & \exists t_2 \leq t + K_k : \neg\ \mathit{wait\ to}\ al_k?\ \mathit{at}\ t_2 \\
\rightarrow & \{C_2^S\} & \exists t_2 \leq t + K_k\, \exists t_0 \leq t_2 + K_S : \mathit{wait\ to\ alarm!\ at}\ t_0\ \mathit{Until\ comm\ via\ alarm} \\
\rightarrow & \{\text{calculus}\} & \exists t_0 \leq t + K_k + K_S : \mathit{wait\ to\ alarm!\ at}\ t_0\ \mathit{Until\ comm\ via\ alarm}
\end{array}$$

Hence the specification of W can be derived provided $K_k + K_S \leq K$.
The processes W_i and S can be implemented as: $W_i \equiv *[re_i? \rightarrow \textbf{skip} \, [] \, \textbf{delay}\ v_i \rightarrow al_i!]$, and $S \equiv [[]_{i=1..n}\, al_i? \rightarrow \mathit{alarm}!]$. With the proof system given in [Hoo89] it can be shown that these programs meet the specifications given earlier, provided: K_S is greater than the maximum duration of a communication via al_i, and K_i is greater than the overhead for a guarded command to start the execution of the delay branch in case of a time-out.

4.3 Watchdog Handlers on a Uniprocessor

In this section we give a possible implementation of the processes P_i. We assume that each P_i is executed on a single processor and that it has to perform a certain task given by a program B_i. When B_i is executing continuously for at least v_i time units this is considered to be an error. For instance, if B_i models a keyboard handler then an error indicates that there is no time left to store characters in a buffer. Such errors can be detected by a watchdog

handler WH_i which executes in parallel with B_i on the same processor and which tries to send signals via channel re_i. We prove that if WH_i has a lower priority than B_i, then WH_i will try to send via re_i at least once every v_i time units iff no error occurs in B_i. Given the specifications for B_i and WH_i below, we show that $\ll$ **prio** 2 (B_i) $|||$ **prio** 1 $(WH_i) \gg$ satisfies the specification of P_i given earlier.

Suppose **prio** 2 (B_i) satisfies the following specification, where the assumption expresses that if any other statement on the same processor executes at a nonscheduling point (with priority ∞), then within a constant time K_c no statement in the environment is executing or some statement executes with a priority lower than 2.

$$\begin{array}{lcl}
(A^{B_i} & \equiv & \mathit{exec}(t, \infty) \rightarrow \exists t_1, t < t_1 \leq t + K_c : \neg \mathit{exec}(t_1, p) \vee (\mathit{exec}(t_1, p) \wedge p < 2), \\
C_1^{B_i} & \equiv & [\mathit{error}_i \rightarrow \exists t\, \forall t_1, t - v_i < t_1 < t : \mathit{exec}(t_1, 2)] \wedge \\
C_2^{B_i} & \equiv & [\exists t\, \forall t_1, t - v_i < t_1 < t : \mathit{exec}(t_1, p) \rightarrow \mathit{error}_i]) : \\
 & & \textbf{prio}\ 2\ (B_i)
\end{array}$$

The specification for the watchdog handler **prio** $1(WH_i)$ commits that it will not execute continuously at nonscheduling points for more than K_c time units (hence satisfying A^{B_i}) and if it waits to communicate, then it waits with priority 1. Finally, if it does not communicate nor waits to communicate via re_i, then it is not executing but requesting processor time.

$$\begin{array}{lcl}
(A^{WH_i} & \equiv & \mathit{true}, \\
C_1^{WH_i} & \equiv & A^{B_i} \wedge \\
C_2^{WH_i} & \equiv & [\mathit{(wait\ to)\ comm\ via}\ re_i!\ \mathit{at}\ t_1 \rightarrow \mathit{(wait\ to)\ comm\ via}\ re_i!\ \mathit{at}\ t_1\ \mathit{with\ prio}\ 1] \wedge \\
C_3^{WH_i} & \equiv & [\neg \mathit{(wait\ to)\ comm\ via}\ re_i!\ \mathit{at}\ t_2 \rightarrow \neg \mathit{exec}(t_2, p) \wedge \mathit{req}(t_2, p)]) : \\
 & & \textbf{prio}\ 1\ (WH_i)
\end{array}$$

To derive the specification for P_i, we first prove

$$\begin{array}{lcl}
(A^i & \equiv & \forall t\, \forall p : \neg \mathit{exec}(t, p), \\
C_1^i & \equiv & [\mathit{error}_i \rightarrow \exists t\, \forall t_1, t - v_i < t_1 < t : \neg \mathit{(wait\ to)\ comm\ via}\ re_i!\ \mathit{at}\ t_1\ ] \wedge \\
C_2^i & \equiv & [\exists t\, \forall t_1, t - v_i < t_1 < t : \neg \mathit{(wait\ to)\ comm\ via}\ re_i!\ \mathit{at}\ t_1\ \rightarrow \\
 & & (\mathit{error}_i \vee \exists t\, \forall t_1, t - v_i < t_1 < t : \neg \mathit{exec}(t_1, p) \wedge \mathit{req}(t_1, p))]) : \\
 & & \textbf{prio}\ 2\ (B_i)\ |||\ \textbf{prio}\ 1\ (WH_i)
\end{array}$$

In this formula, assumption A^i expresses that no other statement is executing on the same processor. To derive this specification, first consider assumption A^{B_i} and prove:
$A^i[e_1/\mathit{exec}] \wedge C_1^{WH_i}[e_2/\mathit{exec}] \wedge (e_1 \vee e_2 \leftrightarrow \mathit{exec}) \rightarrow A^{B_i}$.
From $A^i[e_1/\mathit{exec}]$ which equals $\forall t\, \forall p : \neg e_1(t, p)$, we obtain $e_2 \leftrightarrow \mathit{exec}$. Then it is easy to see that $C_1^{WH_i}[e_2/\mathit{exec}] \wedge (e_2 \leftrightarrow \mathit{exec}) \rightarrow A^{B_i}$. For the commitments we have to prove:
$(C_1^{B_i} \wedge C_2^{B_i})[e_1/\mathit{exec}] \wedge C_2^{WH_i} \wedge C_3^{WH_i}[e_2/\mathit{exec}, r_2/\mathit{req}] \wedge (e_1 \vee e_2 \leftrightarrow \mathit{exec}) \wedge (r_1 \vee r_2 \leftrightarrow \mathit{req}) \rightarrow$
$(C_1^i \wedge C_2^i)$. First consider C_1^i:

$$\begin{array}{lcl}
 & & \mathit{error}_i \\
\rightarrow & \{C_1^{B_i}[e_1/\mathit{exec}]\} & \exists t\, \forall t_1, t - v_i < t_1 < t : e_1(t_1, 2)
\end{array}$$

$$\begin{array}{lcl}
\rightarrow & \{e_1 \rightarrow \mathit{exec}\} & \exists t\, \forall t_1, t - v_i < t_1 < t : \mathit{exec}(t_1, 2) \\
\rightarrow & \{\text{Prio}\} & \exists t\, \forall t_1, t - v_i < t_1 < t : \neg\mathit{(wait\ to)\ comm\ via\ re_i!\ at}\ t_1\ \mathit{with\ prio}\ 1 \\
\rightarrow & \{C_2^{WH_i}\} & \exists t\, \forall t_1, t - v_i < t_1 < t : \neg\mathit{(wait\ to)\ comm\ via\ re_i!\ at}\ t_1
\end{array}$$

Next prove C_2^i:

$$\begin{array}{lcl}
 & & \exists t\, \forall t_1, t - v_i < t_1 < t : \neg\mathit{(wait\ to)\ comm\ via\ re_i!\ at}\ t_1 \\
\rightarrow & \{C_3^{WH_i}[e_2/\mathit{exec}, r_2/\mathit{req}]\} & \exists t\, \forall t_1, t - v_i < t_1 < t : \neg e_2(t_1, p) \wedge r_2(t_1, p) \\
\rightarrow & \{\neg e_2 \rightarrow \neg \mathit{exec} \vee e_1, r_2 \rightarrow \mathit{req}\} & \exists t\, \forall t_1, t - v_i < t_1 < t : (\neg \mathit{exec}(t_1, p) \vee e_1(t_1, p)) \wedge \mathit{req}(t_1, p) \\
\rightarrow & \{\text{calculus}\} & \exists t\, \forall t_1, t - v_i < t_1 < t : e_1(t_1, p) \vee \\
 & & \exists t\, \forall t_1, t - v_i < t_1 < t : (\neg \mathit{exec}(t_1, p) \wedge \mathit{req}(t_1, p)) \\
\rightarrow & \{C_2^{B_i}[e_1/\mathit{exec}]\} & \mathit{error}_i \vee \exists t\, \forall t_1, t - v_i < t_1 < t : \neg \mathit{exec}(t_1, p) \wedge \mathit{req}(t_1, p)
\end{array}$$

Now given the specification for **prio** 2 (B_i) ||| **prio** 1 (WH_i), the rule for $\ll \ldots \gg$-introduction allows us to drop the assumption $A^i \equiv \forall t\, \forall p : \neg \mathit{exec}(t, p)$, because the brackets express that no other processes are executing on the same processor. Furthermore, the commitment is extended with $\forall t\, \forall p : \mathit{req}(t, p) \rightarrow \mathit{exec}(t, p)$ expressing that statements are requesting execution time only if the processor is busy executing an other statement. Observe that with this assertion $\neg \mathit{exec}(t_1, p) \wedge \mathit{req}(t_1, p)$ in C_2^i implies *false*, and thus leads to:
$\exists t\, \forall t_1, t - v_i < t_1 < t : \neg\mathit{(wait\ to)\ comm\ via\ re_i!\ at}\ t_1 \ \rightarrow \mathit{error}_i$
Hence we obtain the following specification for P_i:

$$\begin{array}{rcl}
(A^{P_i} & \equiv & \mathit{true}, \\
C^{P_i} & \equiv & [\mathit{error}_i \leftrightarrow \exists t\, \forall t_1, t - v_i < t_1 < t : \neg\mathit{(wait\ to)\ comm\ via\ re_i!\ at}\ t_1\ ]) : \\
 & & \ll \textbf{prio}\ 2\ (B_i)\ |||\ \textbf{prio}\ 1\ (WH_i) \gg
\end{array}$$

This is exactly the specification which has been used in section 4.1.
It can be shown that **prio** 1 $(*[re_i! \rightarrow \textbf{skip}])$ is a possible implementation of **prio** 1 (WH_i), provided K_c is greater than the maximum duration of a communication via re_i.

5 State of the Art and Conclusion

In particular for concurrent programs communicating via message passing, one can observe a development from non-compositional proof methods which require the (final) program text for their application, such as [AFR80,LG81], towards compositional theories, e.g. [CH81, Sou84,Zwi88] (see [HdR86] for an overview of this development). Whereas these methods verify only safety properties, with linear temporal logic [Pnu77,MP82] also liveness (progress) properties can be verified. Compositional proof systems for temporal logic have been given in [BKP84,NDGO86].

All 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 [Koy89,BH81,SL87] 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 [KVR83] and the verification of local area network protocols [SPE84]. A compositional proof theory for real-time distributed message passing using an assertion language based on real-time temporal logic has been given in [HW89].

Similarly, real-time extensions have been formulated for other methods. Zwarico and Lee [ZL85] have adapted Hoare's trace model [Hoa85] (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 review of formal description techniques for real-time systems can be found in [JG88].

In this paper we have discussed a compositional proof system for real-time distributed message passing in which assumptions can be made about the behaviour of the environment in the style of [MC81,ZREB84]. This formalism has been used for the top-down design of a watchdog timer, thus illustrating the verify-while-develop paradigm. The main idea of the method is that suitable assumptions about the environment reduce the immense number of possible behaviours of complex real-time systems. Although not dealing with real-time, Misra and Chandy [MC81] have used the advantages of assumptions in the hierarchical design and verification of distributed processes with message passing. They proposed a rule for parallel composition and demonstrated their method on several examples. In [ZREB84] these ideas are formalized, resulting in a compositional proof system for assumption/commitment based specifications. The examples in [Oss83] show that the Misra-Chandy method is easy to use and leads to simple and natural correctness proofs. Pandya [Pan88] extends this formalism to asynchronous communication and progress properties.

Our method can be extended to various other communication mechanisms such as several versions of asynchronous communication and broadcast. For instance, we are working on a real-time communication protocol for channels which can loose, reorder and duplicate messages but which satisfy certain real-time constraints such as a maximum message lifetime. We realize, however, that numerous programming language concepts are not treated. This leaves us with a wealth of non-real-time methods which are possible candidates for ramification with real-time.

References

[AFR80] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for Communicating Sequential Processes. *TOPLAS*, 2:359–385, 1980.

[BH81] A. Bernstein and P.K. Harter, Jr. Proving real-time properties of programs with temporal logic. In *Proc. 8th Symposium on Operating System Principles*, pages 1–11, 1981.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. 16th Symposium on Theory of Computing*, pages 51–63, 1984.

[CH81] Zhou Chao Chen and C.A.R. Hoare. Partial correctness of Communicating Sequential Processes. In *Proc. IEEE Int. Conf. on Distr. Computing Systems*, pages 1–12, 1981.

[dR85] W.P. de Roever. The quest for compositionality - a survey of assertion-based proof systems for concurrent programs, Part I: concurrency based on shared variables. In *Proc. IFIP Working Conference 1985: The role of abstract models in computer science*, pages 181–207. North-Holland, 1985.

[HdR86] J. Hooman and W.P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current Trends in Concurrency*, pages 343–395. LNCS 224, 1986.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hoo89] J. Hooman. Compositional specification and verification of distributed real-time systems. Technical report, Eindhoven Univ. of Technology, The Netherlands, 1989.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and Models of Concurrent Systems*, pages 477–498. NATO, ASI-13, 1985.

[HW89] J. Hooman and J. Widom. A temporal-logic based compositional proof system for real-time message passing. In *Parallel Architectures and Languages Europe.* LNCS, 1989.

[JG88] M. Joseph and A. Goswami. Formal description of real-time systems: a review. Research Report RR129, Department of Computer Science, Univ. of Warwick, 1988.

[Koy89] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* PhD thesis, Eindhoven Univ. of Technology, The Netherlands, 1989.

[KSR+88] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79(3):210–256, 1988.

[KVR83] R. Koymans, J. Vytopyl, and W.P. de Roever. Real-time programming and asynchronous message passing. In *Proc. 2nd PODC*, pages 187–197, 1983.

[LG81] G.M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, 15:281–302, 1981.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.

[MP82] Z. Manna and A. Pnueli. Verification of concurrent programs: a temporal proof system. In *Foundations of Computer Science IV, Distributed Systems: Part 2*, volume 159 of *Mathematical Centre Tracts*, pages 163–255, 1982.

[NDGO86] V. Nguyen, A. Demers, D. Gries, and S. Owicki. A model and temporal proof system for networks of processes. *Distributed Computing*, 1(1):7–25, 1986.

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

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319,340, 1976.

[Oss83] M. Ossefort. Correctness proofs of communicating processes: Three illustrative examples from the literature. *TOPLAS*, 5(4):620–640, 1983.

[Pan88] Paritosh Pandya. Compositional verification of distributed programs. Technical Report CS-88/3 (Ph.D. Thesis), Tata Institute of Fundamental Research, Bombay, India, 1988.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th FOCS*, pages 46–57, 1977.

[SL87] A.U. Shankar and S.S. Lam. Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distributed Computing*, 2:61–79, 1987.

[Sou84] N. Soundararajan. Axiomatic semantics for CSP. *TOPLAS*, 6(4):647–662, 1984.

[SPE84] D.E. Shasha, A. Pnueli, and W. Ewald. Temporal verification of carrier-sense local area network protocols. In *Proc. 11th POPL*, pages 54–65, 1984.

[ZL85] A. Zwarico and I. Lee. Proving a network of real-time processes correct. In *Proc. IEEE Real-Time Systems Symposium*, pages 169–177, 1985.

[ZREB84] J. Zwiers, W.P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: soundness and completeness of a proofsystem. Technical Report 57, Univ. of Nijmegen, The Netherlands, 1984.

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes and their relationship.* LNCS 321, 1988.

Session 2:

Process Algebra Applications