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

Citation for published version (APA):

Download date: 18. Apr. 2021
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!eutwsl!wsinjh and mcvax!eutwsl!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).
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 [DG76, 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 can one 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:

\[
\langle S_1 || \cdots || S_m \rangle \rightarrow \cdots \rightarrow \langle S_1 || \cdots || S_m \rangle
\]

where \(\rightarrow\) expresses concurrent execution of the programs \(\langle S_1 || \cdots || S_m \rangle\), each on its own processor. The parallel operator \(||\) expresses that the programs involved are executed on a single processor. The brackets \(\langle\) and \(\rangle\) indicate that no other processes are executed on a particular processor. Actions of parallel processes on a single processor are interleaved, and the basis of priorities assigned to statements. A statement prior \(p\) 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 \( z := e \) 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 \( \langle b_1; IO_1 \rightarrow S_1 \rangle \ldots \langle b_n; IO_n \rightarrow S_n \rangle \rightarrow k; \text{delay} d \rightarrow S \), with \( IO \) 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 \( b \) evaluates to true) can be executed and then continue with the corresponding \( S \). 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 \( [D!e \rightarrow S_1][\text{delay} 5 \rightarrow S_2] \); 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 \( \ast G \) 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 \( \ast G \) 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 formulas 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 \( \langle \rangle \rangle \) 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, Frances 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., \( \{ x = 5 \} \Rightarrow x := x + 1 \{ x = 6 \} \).

First we indicate how a proof system can be formulated to derive such Hoare triples for sequential programs. With \( \{ e \} \) denoting textual substitution of \( e \) for each free occurrence of \( x \) in assertion \( q \), we have the following assignment axiom:

**Axiom 3.1 (assignment)** \( \{ e \} x := e \Rightarrow \{ q \} \)

**Example 3.1** With this axiom we can derive \( \{ x = 5 \} \Rightarrow x := x + 1 \{ x = 6 \} \) because \( x = 6 \) 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)** \( \{ p \} S_{1} \{ q_{1} \}, \{ r \} S_{2} \{ q_{2} \} \Rightarrow \{ p \} S_{1} ; S_{2} \{ q_{1} \} \)

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 pre-conditions and weaken post-conditions, the proof system contains the following rule:

**Rule 3.2 (consequence)** \( \{ p \} \Rightarrow p_{1} \{ q_{1} \}, q_{1} \Rightarrow q \)

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

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

\( \{ true \} B?z \{ x = 3 \} : x := x + 1 \{ x = 4 \} ; E!y(y + 2) \{ x = 4 \} \) and \( \{ y = 3 \} B!y(y + 2) \{ y = 6 \} ; E?y(y + 8) \{ y = 8 \} \).

In this stage only the post-conditions of assignments are verified: from the assignment axiom we obtain \( x = 3 \Rightarrow x := x + 1 \{ x = 4 \} \) and \( y = 6 \Rightarrow y := y + 2 \{ y = 8 \} \). Observe that the post-conditions of the input statements \( B?z \) 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 \( \{ p_{2} \} D!x \{ q_{2} \} \) in the proof outlines of two processes we have to prove \( \{ p_{1} \land p_{2} \} D?x \{ q_{1} \land q_{2} \} \), which is equivalent to \( \{ p_{1} \land p_{2} \} x := e \{ q_{1} \land q_{2} \} \). In our example this leads to the proof obligations:

\( \{ true \land y = 3 \} B?z \{ x = 3 \land y = 3 \} \) and \( \{ y = 6 \land x = 4 \} E?y \{ x + 2 \} \{ y = 6 \land x = 4 \} \) 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

\( \{ p_{1} \land p_{2} \} S_{1} \{ q_{1} \land q_{2} \} \)

**Example 3.2** Consider the parallel composition of \( D!5 \) and \( D?z \), using Hoare triples \( \{ h =<> \} D!5 \{ h =<> \} D?z \{ h =<> \} D?z \{ h =<> \} \).

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 \( \{ h =<> \} D!5 \{ h =<> \} D?z \{ h =<> \} D?z \{ h =<> \} \)

Since the post-condition implies \( v = 5 \Rightarrow z = v \), and hence \( z = 5 \), the consequence rule leads to \( \{ h =<> \} D!5 \{ h =<> \} D?z \{ h =<> \} D?z \{ h =<> \} \).

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** Consider again \( S_{1} \equiv B?z \{ z := x + 1 \} : E!(x + 2) \{ x = 3 \} \) and \( S_{2} \equiv B!y(y + 2) \{ y := y + 2 \} \).

First derive the following Hoare triples:

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

First derive the following Hoare triples:

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

From these the conjunction of the sequential composition rule we obtain:

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

Similarly, \( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

Then the parallel composition rule above leads to

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

The post-condition implies \( v_{1} = 3 \land v_{2} = 1 \land y = v_{1} + 2 \), which leads to

\( \{ p_{1} \land p_{2} \} \)

pre-condition true \( \land y = 3 \) and post-condition \( x = 4 \land y = 8 \) 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., \( \langle (D_{5},S_{1}), (E_{8},S_{1}), (E,0) \rangle \) 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. \( \langle (D,5), (E,6) \rangle \) \( E!6 \{ h =<> \} E!6 \{ h =<> \} \)
- 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, \( \{ h =<> \} D?z \{ 3 \} B?z \{ h =<> \} D?z \{ h =<> \} \)

**Example 3.3** Consider the parallel composition of \( D!5 \) and \( D?z \), using Hoare triples \( \{ h =<> \} D!5 \{ h =<> \} D?z \{ h =<> \} \).

First derive the following Hoare triples:

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

First derive the following Hoare triples:

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

From these the conjunction of the sequential composition rule we obtain:

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

Similarly, \( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

Then the parallel composition rule above leads to

\( \{ h =<> \} B?z \{ h =<> \} B!y(y + 2) \{ y := y + 2 \} \)

The post-condition implies \( v_{1} = 3 \land v_{2} = 1 \land y = v_{1} + 2 \), which leads to

\( \{ p_{1} \land p_{2} \} \)}
\[ z = 4 \land y = 8. \] Thus, by the consequence rule, \( \{ h = \langle \rangle \land y = 3 \} S \parallel S_1 \parallel \{ z = 4 \land y = 8 \}. \) (Again \( h = \langle \rangle \) 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 \( S_1 \equiv B \parallel 10; \ E \parallel D(y) \parallel S_3 \parallel D(t+z) \parallel E(z+1). \) Following the previous example, we could first prove:

- \( \{ h = \langle \rangle \} S_1 \{ q_1 \equiv \exists ! t : h = \langle (B,0),(E,v_1) \rangle \land x = v_1 \}, \)
- \( \{ h = \langle \rangle \} S_2 \{ q_2 \equiv \exists ! t : h = \langle (B,v_2),(D,v_2+1) \rangle \land y = v_2 \}. \) 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. 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 \( h_{\text{cset}} \), denotes the sequence obtained from the history denoted by \( h \) by removing all records with channel name not in \( cset \).

For instance, \( h = \langle (B,0),(D,1),(E,3) \rangle \), and 
- \( h_{\{B\}} = \langle (B,0) \rangle \),
- \( h_{\{D\}} = \langle (D,1) \rangle \),
- \( h_{\{E\}} = \langle (E,3) \rangle \),
- \( h_{\{B,D\}} = \langle (B,0),(D,1) \rangle \),
- \( h_{\{B,E\}} = \langle (B,0),(E,3) \rangle \),
- \( h_{\{D,E\}} = \langle (D,1),(E,3) \rangle \),
- \( h_{\{B,D,E\}} = \langle (B,0),(D,1),(E,3) \rangle \).

At parallel composition of \( S_1 \) and \( S_2 \) we require that the post-condition of \( S_1 \) 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)**

\[
\begin{align*}
\{ p_1 & \} S_1 \{ q_1 \} , \{ p_2 & \} S_2 \{ q_2 \} \quad \{ p_1 \land p_2 \} S_1 \parallel S_2 \{ q_1 \land q_2 \}
\end{align*}
\]

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 \( S_2 \). Recall that Hoare triples are not sufficient to formulate the cooperation test but that proof obligations 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 \( S_1 \parallel S_2 \). 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 \( p \rightarrow p_1 \land p_2 \), \( q_1 \land q_2 \rightarrow q \) 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 formulate \( (A,C) : \{ p \} S \{ q \} \), where

- \( A \) is an assumption describing the expected behaviour of the environment of \( S \),
- \( C \) 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, \( seq_1 \sqsubseteq seq_2 \) 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, \( h_D \sqsubseteq \langle (D,3) \rangle \) which expresses that the history of channel \( D \) is a prefix of \( \langle (D,3) \rangle \), can be used as an assumption as follows:

- \( h_D \sqsubseteq \langle (D,3) \rangle \), \( h_B \sqsubseteq \langle (B,4) \rangle \), \( \{ \text{true} \} D \mid z \{ x = 3 \} \).

This assumption can be used for a commitment about the next communication:

- \( h_D \sqsubseteq \langle (D,3) \rangle \), \( h_B \sqsubseteq \langle (B,4) \rangle \), \( \{ \text{true} \} D \mid z \{ x = 3 \} \).

A proof system for these assumption-commitment based formulae has been given in [Ho93]. In this paper we discuss mainly the proof obligations for assumptions and commitments at parallel composition. Consider the parallel composition \( S_1 \parallel S_2 \) 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 \( S_1 \parallel S_2 \)? 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 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 $S_1 || S_2$.

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

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 $A \equiv \top$ lead to circular reasoning, e.g. $A_1 \rightarrow C_1 \rightarrow A_2 \rightarrow C_1 \rightarrow A_1$. 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 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:

$$A \land C_1 \rightarrow A_2, A \land C_2 \rightarrow A_1 = (A \land C_1 \land C_2) : (p_1, p_2) S_1 \{q_1 \land q_2\}$$

**Rule 3.4 (par. comp. A-C)**

Example 3.6 Consider $S_1 \equiv Bx \mid x := x + 1; E(x + 2) \mid S_2 \equiv D'y; By; B'y; y := y + 2$.

Then for $S_1$ and $S_2$ we can derive

$$(A_1 \equiv h_B \leq (B, 3) \land C_1 \equiv h_B \leq (E, 6)) \Rightarrow \{h_B \leftarrow c\} S_1 \{x = 4\}$$

and

$$(A_2 \equiv h_B \leq (B, 3) \land C_2 \equiv h_B \leq (E, 6)) \Rightarrow \{h_B \leftarrow c\} S_2 \{y = 8\}$$

Since $S_1$ and $S_2$ communicate with each other via the channels $B$ and $E$, we take for $S_1 \mid S_2$ the assumption about the remaining channel: $A \equiv h_B \leq (D, 3) >$. Then

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

$$(A, C_1 \land C_2) : \{h_B \leftarrow c\} S_1 \{q_1 \land q_2\} = (A, C_1 \land C_2) : \{h_B \leftarrow c\} S_2 \{x = 4 \land y = 8\}.$$  

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 $t$ is introduced which refers to this external clock. E.g. if an assignment takes between 2 and 5 time units, then \(x = 5 \land time = 2\) $x := x + 1 \{x = 6 \land 5 < time < 7\}$. 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 each channel $D$:

**Axiom 3.2 (MP)** $∀t: \neg(wait \ to \ D? \ at \ t \ \land \ wait \ to \ D! \ at \ t)$

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)** $∀t: \neg((wait \ to \ D? \ at \ t \ \lor \ wait \ to \ D! \ at \ t) \land \ comm \ via \ D \ at \ t)$

The following abbreviations are frequently used:

- $comm \ via \ D \ at \ t \ \equiv \exists v: \ comm \ via \ D \ at \ t \ with \ value \ v$
- $(\text{wait to}) \ comm \ via \ D! \ at \ t \ \equiv \ \text{comm} \ via \ D \ at \ t \ \lor \ \text{wait to} \ D! \ at \ t$
- $wait \ to \ D! \ during (t_1, t_2) \equiv \ \text{wait to} \ D! \ at \ t_1 \ \land \ \text{wait to} \ D! \ at \ t_2$
- $wait \ to \ D! \ at \ t \ \land \ Until \ comm \ via \ D \ \equiv \ \text{wait to} \ D! \ at \ t \ \lor \ \exists t_1 \ t_1 < t_2: \ \text{wait to} \ D! \ at \ t_1 \ \land \ \text{comm} \ via \ D \ at \ t_2$

From these definitions we derive the following lemma.

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

**Example 3.7** Consider processes $P_1$ and $P_2$ with specifications

$\{p\} P_1 \{\text{wait to} \ D! \ at \ 5 \ \land \ \text{Until comm} \ via \ D\}$ and

$\{p\} P_2 \{\text{wait to} \ D! \ at \ 5 \ \land \ \text{Until comm} \ via \ D\}$.

Then the rule for parallel composition leads to the following post condition for $P_1 \mid P_2$:

$\text{wait to} \ D! \ from \ 5 \ \land \ \text{Until comm} \ via \ D \ \land \ \text{wait to} \ D! \ from \ 5 \ \land \ \text{Until comm} \ via \ D$.

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:

- $exec(t, c)$ which is true when the process is executing at a non-scheduling point $t$.
- $exec(t, p)$, for $p \neq \infty$, which is true when the process is executing at a scheduling point $t$ with priority $p$.
- $req(t, p)$ 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)** \( \text{exec}(t, p_1) \land p_1 > p_2 \rightarrow (\text{wait to}) \text{comm via D}! at t with prio p_2 \)

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):

\[
\begin{align*}
A & \equiv (\text{wait to}) \text{comm via D}! at 5 \land \forall t, 3 \leq t < 5: \neg \text{comm via D}! at t, \\
C & \equiv \forall t, 3 \leq t < 5: \text{wait to D}! at t \land \forall t, 5 < t < 6: \text{comm via D}! at t) : \{\text{time}=3\} D? \{\text{time}=6\}
\end{align*}
\]

Note that, due to the maximal progress constraint, a communication takes place as soon as the environment of A is 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 \( q_1 \land q_2 \) could imply 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:

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):

\[
\begin{align*}
(A_1 & \equiv (\text{wait to}) \text{comm via B}! at 2 \land \forall t, 0 \leq t < 2: \neg \text{comm via B}! at t \land \\
C_1 & \equiv \forall t, 2 < t < 3: \text{comm via B}! at t \land \text{wait to D}! at 3 \land \text{Until comm via D}! \land \\
& \{\text{time}=0\} S_1 \equiv \ll B! 11; D! 15; \text{action 2} \gg \{\text{time}=9\}, \text{and} \\
(A_2 & \equiv \text{wait to D}! at 3 \land \text{Until comm via D}! \land \\
C_2 & \equiv \{\text{time}=0\} S_2 \equiv \ll B! 11; D! 15; \text{action 2} \gg \{\text{time}=9\})
\end{align*}
\]

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

For the unprocessor 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 \( S_1 \parallel S_2 \) executes iff \( S_1 \) or \( S_2 \) executes, we add the property \( \neg \text{exec} \leftrightarrow e_1 \land e_2 \), which is an abbreviation of \( \forall p \forall t: \text{exec}(t, p) \rightarrow e_1(t, p) \land e_2(t, p) \).

Similar changes are made for the req predicate. This leads to the following conditions for deriving a pair \((A, C)\) for \( S_1 \parallel S_2 \) from pairs \((A_1, C_1)\) for \( S_1 \) and \((A_2, C_2)\) for \( S_2 \):

\[
\begin{align*}
A & \equiv \ll e_1 / e_2; r_1 / r_2 \rr \land \ll e_1 / e_2; r_1 / r_2 \rr \land \{e_1 \land e_2 \rightarrow \neg \text{exec}\} \land \{r_1 \lor r_2 \rightarrow \neg \text{req}\} \rightarrow A_1, \text{and} \\
C & \equiv \ll e_1 / e_2; r_1 / r_2 \rr \land \ll e_1 / e_2; r_1 / r_2 \rr \land \{e_1 \land e_2 \rightarrow \neg \text{exec}\} \land \{r_1 \lor r_2 \rightarrow \neg \text{req}\} \rightarrow C
\end{align*}
\]

### 4 Example Watchdog Timer

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

![Watchdog Timer Network](image_url)

\( W \) 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 \( r_{ei} \) at least every \( v_i \) (say) time units. Process \( W \) is always ready to receive signals on any \( r_{ei} \). If there is no communication on a particular \( r_{ei} \) for \( v_i \) 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 \( P_1 \parallel \cdots \parallel P_n \parallel W \) that one of the \( P_i \) is not functioning correctly 
iff \( W \) tries to send a 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 for \( W \). 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 \( \ll \text{prio} 2 (B_i) \rr \ll \text{prio} 1 (W_i) \rr \) where \( B_i \) performs the real task of \( P_i \) and \( W_i \) is a watchdog handler which sends signal via \( r_{ei} \) as long as \( B_i \) is functioning properly.

In this example we concentrate on the assumption-commitment reasoning. Henceforth all specifications have pre-condition \( \text{time}=0 \) and post-condition \( \text{true} \).
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 \texttt{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 \texttt{rec}. Furthermore, if there is a waiting period for input via \texttt{rec} of \( v_k \) time units, then, for some constant \( K \), \( W \) starts waiting to output on channel \texttt{alarm} until the actual communication takes place. Hence we specify \( W \) as follows.

\[
\begin{align*}
(A^W & \equiv \text{true,} \\
C_0^W & \equiv \text{[wait to rec where } (t_2 - v_k, t_2) \rightarrow } \\
C_1^W & \equiv \text{[(wait to) comm via alarm] at } t_1 \rightarrow \\
C_2^W & \equiv \text{[(wait to) comm via alarm] at } t_1 \rightarrow \\
\end{align*}
\]

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 \texttt{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 \texttt{error}, which represents some erroneous behaviour of \( P_i \). For all \( i \):

\[
\begin{align*}
(A^P_i & \equiv \text{true,} \\
C^P_i & \equiv \text{[error} \rightarrow \exists t: (wait to) comm via rec_1 at } t_1 \] : P_i
\end{align*}
\]

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 either communicating via \texttt{rec}, nor waiting to communicate via \texttt{rec}. Given our specifications for \( P_1, \ldots, P_n \) and \( W \), we try to prove that \( P_1 \parallel \cdots \parallel P_n \parallel W \) has commitment \( \exists k: \text{error} \rightarrow \exists t: (\text{wait to}) \text{ comm via alarm} at t \). We can apply parallel composition \( n \) times and obtain (true, \( C_1^P \land C_2^P \land \ldots \land C_n^P \land C^W \)) : \( P_1 \parallel \cdots \parallel P_n \parallel W \ldots \) 

First prove \( \exists k: \text{error} \rightarrow \exists t: (\text{wait to}) \text{ comm via alarm} at t \).

\[
\exists t: (\text{wait to}) \text{ comm via alarm} at t \rightarrow \\
\{C^W\} \exists t: \exists t_0 \leq t: \text{ wait to rec at } (t_0 - v_k, t_0) \\
\{\text{definition}\} \exists t_0 \exists t_0 \geq t_0 < t: \text{ wait to rec at } t_1 \\
\{\text{MP, Excl}\} \exists t_0 \exists t, t_0 < t_0 < t_0 \leq t_1: \text{ wait to rec at } t_1 \land \text{ comm via rec at } t_1 \\
\{\text{definition}\} \exists t_0 \exists t, t_0 < t_0 < t_0 \leq t_1: \text{ wait to rec at } t_1 \land \text{ comm via rec at } t_1 \\
\{C^P\} \exists k: \text{error}
\]

Next we try to prove \( \exists k: \text{error} \rightarrow \exists t: (\text{wait to}) \text{ comm via alarm} at t \).

From \( \exists k: \text{error} \) we obtain, by \( C^P \), \( \exists t \exists t_0 \exists t_1, t_0 < t_0 < t_1: (\text{wait to}) \text{ comm via rec at } t_1 \). Thus \( \exists k \exists t \exists t_0 \exists t_1, t_0 < t_0 < t_1: (\text{wait to}) \text{ comm via rec at } t_1 \). This 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 \texttt{skip} would satisfy its specification. Therefore we modify the specification for \( W \) as follows:

\[
\begin{align*}
(A^W & \equiv \text{true,} \\
C_0^W & \equiv \text{[(wait to) comm via alarm] at } t_1 \rightarrow \\
C_1^W & \equiv \text{[(wait to) comm via alarm] at } t_1 \rightarrow \\
C_2^W & \equiv \text{[(wait to) comm via alarm] at } t_1 \rightarrow \\
\end{align*}
\]

Note that \( C_0^W \) follows from \( C_0^P \). Now we prove the following steps as follows for all \( k \), \( \text{error} \):

1. \( \{C^P\} \exists t_1, t_0 < t_0 < t: (\text{wait to}) \text{ comm via rec at } t_1 \\
2. \{\text{definition}\} \exists t_1, t_0 < t_0 < t: (\text{comm via rec at } t_1 \\
3. \{C^P\} \exists k: \text{error} \rightarrow \exists t_1, t_0 < t_0 < t: (\text{wait to}) \text{ comm via alarm at } t_1 \\
4. \{\text{Lemma 3.1}\} \exists t_0: (\text{wait to}) \text{ comm via alarm at } t_0

This reasoning allows us to verify properties of \( P_1 \parallel P_2 \parallel \cdots \parallel P_n \parallel W \) (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, \( W \equiv P_1 \parallel \cdots \parallel P_n \parallel S \), where each \( P_i \) is a watchdog timer for process \( P_i \). \( W \) signals process \( S \) via channel \texttt{al}, as soon as there is no communication on \texttt{rec} for at least \( v_i \) time units. Process \( S \) waits for a signal on any of the \texttt{al},s; after receipt of a signal it tries to send a message on \texttt{alarm} (see Fig. 2).

![Figure 2: Implementation Watchdog Timer](image-url)
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 $a_k$. If $S$ does not try to communicate nor communicates via one of the $a_k$ 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.

\[
(A_S^g \equiv \text{true},
C_S^g \equiv ([\text{wait to}] \text{comm via alarm}) \text{ at } t_1 \rightarrow \exists k \exists t_0 \leq t_1 : \text{comm via } a_k \text{ at } t_0 \land 
C_S^g \equiv [\neg \text{ wait to } a_k] \text{ at } t_2 \rightarrow 
3t_0 \leq t_2 + K_S : \text{wait to alarm! at } t_0 \text{ Until comm via alarm}) : S
\]

We show that $W_1 \mid \cdots \mid W_m S$ satisfies the specification of $W$. We can take, by repeated application of the parallel composition rule, the conjunction of commitments: $A^p_1 (C^p_1) \land C^p_2 \land C^p_3$. This implies $C^p_2$ as follows:

\[
\begin{align*}
& (\text{wait to}) \text{comm via alarm! at } t_1 \\
& \rightarrow \{C^p_2\} \exists k \exists t_0 \leq t_1 : \text{comm via } a_k \text{ at } t_0 \\
& \rightarrow \{\text{definition}\} \exists k \exists t_0 \leq t_1 : (\text{wait to}) \text{comm via } a_k \text{ at } t_0 \\
& \rightarrow \{C^p_1\} \exists k \exists t_0 \exists t_2 \leq t_0 : \text{wait to req? during } (t_0 - v_k, t_0) \\
& \rightarrow \{t_0 \leq t \wedge t_2 \leq t_1\} \exists k \exists t_0 \leq t_1 : \text{wait to req? during } (t_0 - v_k, t_0)
\end{align*}
\]

Next we prove $C^p_2$:

\[
\begin{align*}
& \exists t Vt_1, t - v_k < t_1 < t \rightarrow \neg \text{comm via } a_k \text{ at } t_1 \\
& \rightarrow \{C^p_2\} \exists k \exists t_0 \leq t + K_K : (\text{wait to}) \text{comm via } a_k \text{ at } t_2 \\
& \rightarrow \{\text{definition}\} \exists k \exists t_0 \leq t + K_K : \text{wait to req? during } (t_0 - v_k, t_0) \\
& \rightarrow \{\text{MP, Excl}\} \exists k \exists t_0 \leq t + K_K : \neg \text{wait to req? at } t_2 \\
& \rightarrow \{C^p_1\} \exists k \exists t_0 \leq t + K_K : \text{wait to alarm! at } t_0 \text{ Until comm via alarm} \\
& \rightarrow \{\text{calculus}\} \exists k \exists t_0 \leq t + K_K : \text{wait to alarm! at } t_0 \text{ Until comm via alarm}
\end{align*}
\]

Hence the specification of $W$ can be derived provided $K_K + K_K \leq K$.

The processes $W_i$ and $S$ can be implemented as: $W_i \equiv \{r_k \rightarrow \text{skip} : \text{delay } v_i \rightarrow a_i[l]\}$, and $S \equiv [[V_{a_k} : a_k] \rightarrow \text{alarm}]$. 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 $a_k$, and $K_K$ 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 $W_H$, which executes in parallel with $B_i$ on the same processor and which tries to send signals via channel $r_k$. We prove that if $W_H$ has a lower priority than $B_i$, then $W_H$ will try to send via $r_k$ at least once every $v_i$ time units iff no error occurs in $B_i$. Given the specifications for $B_i$ and $W_H$ below, we show that $\langle \text{prio } 2 (B_i) \mid \mid \text{prio } 1 (W_H) \rangle$ satisfies the specification of $B_i$ given earlier.

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

\[
(A^B_i \equiv \text{exec(t, } \infty) \rightarrow \exists t_0, t < t_1 < t + K_e : \neg \text{exec(t, } p) \lor \text{exec(t, } p) \land p < 2), \\
C^B_i \equiv \{\text{error} \rightarrow \exists t Vt_1, t - v_i < t_1 < t : \text{exec(t, } p) \lor \text{exec(t, } p) \land p < 2)\}
\]

The specification for the watchdog handler $\text{prio } 1 (W_H)$ commits it not to execute continuously at nonscheduling points for more than $K_e$ 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 $r_k$, then it is not executing but requesting processor time.

\[
(A^{W_H} \equiv \text{true}, \\
C^{W_H} \equiv A^B_i \land \\
C^{W_H} \equiv [[\text{wait to}] \text{comm via } r_k \rightarrow \text{wait to}] \text{comm via } r_k \rightarrow \neg \text{exec(t, } p) \lor \text{exec(t, } p) \land p < 2) : \text{prio } 1 (W_H)
\]

To derive the specification for $P_i$, we first prove

\[
(A' \equiv \forall Vt Vp : \neg \text{exec(t, } p), \\
C'_1 \equiv \{\text{error} \rightarrow \exists t Vt_1, t - v_i < t_1 < t : \neg \text{exec(t, } p) \lor \text{exec(t, } p) \land p < 2) \land \\
C'_2 \equiv \{\text{error} \rightarrow \exists t Vt_1, t - v_i < t_1 < t : \neg \text{exec(t, } p) \lor \text{exec(t, } p) \land p < 2) : \text{prio } 2 (B_i) \mid \mid \text{prio } 1 (W_H)
\]

In this formula, assumption $A'$ expresses that no other statement is executing on the same processor. To derive this specification, first consider assumption $A^B_i$ and prove:

\[
A^i(e_1 : \text{exec}) \land \text{C}^{W_H}_i(e_2 : \text{exec}) \land (e_1 \lor e_2 \rightarrow \text{exec}) \rightarrow A^B_i.
\]

From $A^i(e_1 : \text{exec})$ which equals $\forall Vt : \neg \text{exec(t, } p)$, we obtain $e_2 \rightarrow \text{exec}$. Then it is easy to see that $\text{C}^{W_H}_i(e_1 : \text{exec}) \land (e_2 \rightarrow \text{exec}) \rightarrow A^B_i$. For the commitments we have to prove:

\[
\text{C}^{B_i}_1 \land \text{C}^{B_i}_2 \equiv \{\text{error} \rightarrow \exists t Vt_1, t - v_i < t_1 < t : \neg \text{exec(t, } p) \lor \text{exec(t, } p) \land p < 2) \rightarrow (C^i_1 \land C^i_2)
\]

First consider $C^i_1$:

\[
\text{error}; \\
\rightarrow \{C^i_1(e_1 : \text{exec})\} \exists t Vt_1, t - v_i < t_1 < t : e_1(t, t_2)
\]
→ \{e_1 \rightarrow exec\} \ \exists t, v_1 < t \leq t : exec(t, 2)
→ \{Prio\} \ \exists t, v_1 < t < t : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t \ \text{with} \ t \ \text{prio 1}
→ \{C^{WH}_{e}(r_2|\text{exec}, r_2|\text{req})\} \ \exists t, v_1 < t < t : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t
→ \{C^{WH}_{e}(\text{exec}, r_2|\text{exec}, r_2|\text{req})\} \ \exists t, v_1 < t < t : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t
Next prove \text{C}_1:

\forall t_1, t_2, t_3, v_1, v_2, v_3 : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t_1
\rightarrow \text{exec}_{v_1} \ \exists t_4, t_5 : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t_2
\rightarrow \text{exec}_{v_2} \ \exists t_4, t_5 : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t_3

Now given the specification for \text{prio 2} (B_1) \ \| \ \text{prio 1} (WH_1), the rule for \text{||} \ \ldots \ \text{||} \ \text{protection allows us to drop the assumption} \ A' \ \\forall t, v_1 : \neg \text{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, v_1 : \neg \text{exec}(t, p) \rightarrow \text{exec}(t, p) expressing that statements are requesting execution only if the processor is busy executing an other statement. Observe that with this assertion \text{exec}(t, p) \ \text{implies} \ \text{false}, and thus leads to:

\forall t_1, t_2, v_1, v_2, v_3 \leq t : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t_1 \rightarrow \text{error}

Hence we obtain the following specification for \text{P_1}:

\text{(A^{P_1}_{\text{exec}}) } \equiv \text{true,}
\text{C^{P_1}_{\text{exec}} } \equiv \text{[error, } \rightarrow \exists t_1, t_2, v_1, v_2, v_3 \leq t : \neg \text{(wait to)} \ \text{comm via} \ \text{req} \ \text{at} \ t_1] : \ \text{||} \ \text{prio 1} (WH_1) \ \Rightarrow

This is exactly the specification which has been used in section 4.1.

It can be shown that \text{prio 1} (\text{exec} \ \text{exec} \ \text{exec} \ \text{exec}) is a possible implementation of \text{prio 1} (WH_1), provided \text{K_e} is greater than the maximum duration of a communication via \text{req}.

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 \text{[AFR80, LG81]}, towards compositional theories, e.g. \text{[CHS1, Sou84, W888]} (see \text{[HD86]} for an overview of this development). Whereas these methods verify only safety properties, with linear temporal logic \text{[Pnu77, MP82]} also liveliness (progress) properties can be verified. Compositional proof systems for temporal logic have been given in \text{[BKP84, NDG86]}. 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 \text{[Koy98, BH81, SL97]} 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 \text{[KVR83]} and the verification of local area network protocols \text{[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 \text{[HWW89]}.

Similarly, real-time extensions have been formulated for other methods. Zwarico and Lee \text{[ZL85]} have adapted Hoare’s trace model \text{[Boa85]} (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 \text{[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 \text{[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 \text{[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 \text{[ZREB84]} these ideas are formalized, resulting in a compositional proof system for assumption/commitment based specifications. The examples in \text{[Oss83]} show that the Misra-Chandy method is easy to use and leads to simple and natural correctness proofs. Pandya \text{[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


Session 2:
Process Algebra
Applications