Proving monitors revisited: a first step towards verifying object oriented systems
Gerth, R.T.; de Roever, W.P.

Published: 01/01/1986

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 author’s 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

Citation for published version (APA):

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?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 21. Dec. 2018
PROVING MONITORS REVISITED

Rob Gerth
Willem P de Roever

August, 1986
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.
PROVING MONITORS REVISITED:

a first step towards verifying object oriented systems*

Rob Gerth1,2 and W. P. de Roever2
Eindhoven University of Technology3

August 1986

Abstract. An axiomatic characterization of monitors, based on assumption-commitment style reasoning, is given that is sound and (relatively) complete. This characterization is based on the fundamental notions of cooperation and interference, but does not use them as second order notions. The cooperation test was originally conceived to capture the proof theoretical analogue of distributed message passing between disjoint processes, as opposed to the interference freedom test, being the proof theoretical analogue of concurrency based on interference by jointly shared variables. Since then, the cooperation test has been applied to characterize synchronous communication in Hoare’s Communicating Sequential Processes, Ichbia’s Ada, and Wirth’s Modula-2, supported by soundness and completeness proofs. An overview is given of the rationale underlying this characterization, culminating in the development of proof systems for a new monitor based programming language for concurrency (Communicating Modules, CM) which combines distributed message passing between processes with interference through local variables of a process which are shared between its sub-processes. As such this is a first step towards the formal verification of object oriented systems. In this context, we also show how the method, traditionally caugh in terms of proof outlines, can be rendered syntax directed in a precise and formal way. In a separate paper, the proof system has been shown to be sound and (relatively) complete.


1the author was partially supported by the Foundation for computer science research in the Netherlands (SION) with financial aid from the Netherlands organization for the advancement of pure research (ZWO).

2the author is currently working in and partially supported by ESPRIT project 937: “Debugging and Specification of Ada Real-Time Embedded Systems (DESCARTES)”.

1. INTRODUCTION

The notion of monitor originates from C.A.R. Hoare and P. Brinch Hansen [Hoa74, BH75] in their work on managing resources in operating systems. The concept proved to be a healthy one as witnessed by the programming languages that are based on their idea: Concurrent Pascal [BH75], Distributed Processes (DP) [BH78], Mesa [MMS79] and, most recently, Wirth's Modula (-2) [W84]. Also, the Ada rendezvous mechanism derives from it [ARM83]. In a different area, the concept surfaces in object oriented systems [Am86].

This widespread use of the monitor concept provides ample motivation for investigating its proof theoretical characterization. This will be done in the context of the (new) language Communicating Modules (CM), which incorporates the bare essentials of the monitor concept.

The proof system is based on the fundamental notions of cooperation and interference freedom. One of our aims is finding out to what extend these notions support truly syntax-directed proof systems. This results in an assumption-commitment style system that contrasts sharply with the more traditional formulations.

The cooperation test of Apt, Francez and de Roever [AFdeR80] was originally conceived to capture the proof theoretical analogue of distributed message exchange between disjoint processes, as opposed to Owicki and Gries' interference freedom test [OG76], being the proof theoretical analogue of concurrent communication by means of interference through jointly shared variables. Some authors ([ILG81, LS84, SS84]) stress that both forms of communication can be proof theoretically characterized using interference freedom only, since proofs for both ultimately amount to an invariance proof of one global assertion [Ash75], invariance of whose parts amounts to interference freedom. In some sense this is a truism, because all such "syntax-directed" methods can be seen as syntactic elaborations of Floyd's inductive assertion method, which can be formulated as finding an invariant that should be maintained (see Lamport in [AFdeR80] and [G84]); in this sense there is nothing new under the sun. Our view is that notions of interference freedom and cooperation derive their importance on the level of methodology; namely from the way in which these methods (implicitly) analyse concurrency. In that sense, we maintain that the notion of cooperation is geared to communication-based concurrency, more than interference freedom is. This thesis is supported by the smooth way ([HdeR86]) in which cooperation based proof systems can be transformed into the simple and elegant compositional proof systems of [MC81] and [ZdeRB85] and by the assumption-commitment style formulation of our proof system in section 5. Also, we feel that the characteristic nature of the cooperation test is still preserved in the analysis of these authors, because in their analysis of CSP the part dealing with interference freedom specializes to maintenance of a global invariant, the expression of which requires per process the introduction of auxiliary variables which are updated in that process only, thus preserving the concept of disjointness (as opposed to sharing), since now all variables from different processes are disjoint. This point is analyzed in detail in [HdeR86].

The cooperation test has been applied to characterize concurrent communication as occurring in Hoare's Communicating Sequential Processes (CSP) [AFdeR80], Ichbiah's Ada [GdeR84], and Brinch Hansen's Distributed Processes (DP)
[GdeRR82a]. This characterization has been certified through soundness and completeness proofs [A83, G86]. As in the interference freedom test this characterization consists of two stages, a local sequential stage and a global stage: In the local stage one constructs purely sequential proof outlines for the component processes in isolation, containing assumptions about the communication actions. Proof outlines associate every statement with appropriate pre- and postconditions, such that Hoare's axioms and rules for sequential programming constructs are satisfied. In the global stage these proof outlines are checked for mutual consistency through a cooperation test. This is needed, as communication between processes forces proof outlines to make assumptions about the behaviour of the environment of a process. The interaction between processes is captured by a general invariant, GI, which keeps track of the values communicated between processes during their execution. The expression of GI may require introduction of auxiliary quantities called auxiliary variables (also called mathematical, logical, ghost or fanthom variables by other authors). Since these auxiliary variables must be updated when communication occurs, GI may be temporarily violated. Therefore bracketed sections are introduced to confine the assignments to the free variables of GI syntactically. By postulating that these variables are updated in bracketed sections only, once invariance of GI over all bracketed sections has been established, it suffices to assume GI in the precondition of a concurrent program in order to conclude GI in its postcondition. (Of course the introduction of these auxiliary quantities is paralleled by their elimination via various auxiliary variable elimination rules, auxiliary variable substitution rules, and parameter substitution rules. This is the subject of [A81], and is not stressed in the present paper; consult [deR85] for the foundations of the link between concurrency and proof theory.)

The present paper stresses the uniform nature of the strategy outlined above. First the cooperation test is reviewed in its original form for CSP, in section 2. Then it is combined with the notion of a canonical proof outline for procedure bodies (originally due to [Hoa71]) in order to capture the mechanism of remote procedure calls. On account of the possibility of occurrences of these calls inside procedure bodies, this canonical nature of proof outlines is mandatory, since, otherwise, every entry call or rendezvous would require a new proof outline for the appropriately modified procedure body, which, in case it contains in its turn a call, would require another proof outline for this call, etc.

Also, the possibility of having procedure calls within procedure bodies calls for a refinement of the notion of bracketed section, since at such a nested call new communication will occur and, therefore, GI has to hold again, implying that a nested call should not occur inside a bracketed section. This is the subject of section 3, where a proof theoretical characterization of the Ada rendezvous concept is reviewed.

Section 4 contains an overview of the interference freedom test of [OG76] since this test has to be incorporated in the proof system for Communicating Modules in section 5.

In monitor based systems, distributed message exchange between processes through remote procedure calls is combined with an interleaving-type of execution within the processes. The (sub) processes whose executions are interleaved, consist of (active) instances of entry procedures, who share the monitor's variables.
Owicki's concept of interference freedom is now taken over, but has to be generalized, because:

- an atomic action of an (instance of an) entry procedure now extends from one programmer defined synchronization point, at which the procedure acquires the monitor lock, to another, at which it relinquishes it. Such actions are therefore not anymore syntactically determined as in Owicki's case, but are semantically determined,
- the number of subprocesses active at a certain time within a process is dynamically varying, corresponding as it does with the number of executed but not yet completed entry calls from outside that process at that time,
- accepting an entry call results in a new procedure instance being created and executed.

This results in an interleaving mode of execution of (the atomic actions of) these subprocesses.

Now our introducing canonical proof outlines for procedure bodies comes to good stead. For the fact that the proof of an entry procedure is canonical implies that this proof characterizes the behaviour of each instance, implying that also interference freedom tests can be made canonical in principle. As in the usual interference freedom test, the coexistence of various instances inside a process at specific locations is expressed by the conjunction of the assertions attached to those locations in the various proof outlines, thus enabling the expression of the precondition pre of an interference freedom test, since pre expresses the dynamic coexistence of procedure instances inside a process. Instance counters have to be wired into the proof system to express the coexistence of instances of the same procedure. Finally, the fact that procedure bodies are canonical also implies that the locations at which our grains of interleaving (atomic actions) may start or end due to synchronization, i.e. the monitor lock being on or off, can be syntactically characterized in canonical fashion. This happens by combining in these proof outlines assertions expressing whether the monitor lock is on or off with the conjunction of (i) the assertion to be proved invariant and (ii) the above assertion pre which expresses coexistence of the respective instances.

This idea of viewing a monitor as a shared variable concurrent program and, consequently, of introducing the concept of interference freedom, is missing in previous attempts to devise proof rules for monitor based languages [Hoa74, How76, AB81] and this is what makes their proof systems incomplete in the technical sense of the word.

The above discussion is couched in terms of proof outlines and the cooperation and interference freedom test are tests on proof outlines. In fact, the traditional formulation of proof systems based on these notions, [OG76, AFdeR80, LG81, GdeR84], might better be said to yield verification algorithms, since the formulation of the proof rules is not always in terms of the specifications to be proved; see, e.g., the parallel composition rule in section 2. The proof system of section 5 shows that proof systems in the formal sense of the word can be based on the notions of cooperation and interference. However, for this the notion of specification has to be extended. Section 5.2 develops and motivates the required extension. In section 5.3, the proof system is developed and section 5.4 contains a
small example proof. Soundness and relative completeness for this proof system has been shown in [G86].

As is well-known, ours is not the only way to achieve syntax-directedness for cooperation and interference freedom style reasoning. K. R. Apt showed for CSP ([A84]) and, more recently, C. Stirling showed for shared variable concurrency ([St86]), how syntax-directedness can be achieved in this context through a change in the notion of deduction. To wit, they use a Gentzen style sequent calculus instead of Hilbert style reasoning. While interesting in their own right, we feel that these results somewhat miss the point. Non-syntax directedness reflects the weakness of partial correctness specifications for concurrent programs: such specifications fail to specify the component interactions during computations. To us, the interesting question is what specifications do support syntax directedness — and Hilbert style — proofs, rather than what notion of deduction is needed to formulate a formally correct proof system based on partial correctness specifications.

Finally, we mention the work of A. Sobel and N. Soundararajan [SS85]. They give a syntax directed axiomatization for partial correctness properties of Brinch Hansen’s Distributed Processes (DP), using the Soundararajan-Dahl approach [So84]. Basically, a linear history semantics for DP is lifted to the level of proof. The semantic consistency checks on the histories when composing the denotations of a DP process or program are mirrored in the proof system by a CONSIST and a COMPAT predicate. These predicates are the counterpart of our interference freedom and cooperation tests.

In section 6, we reflect on the obtained results.

2. A PROOF SYSTEM FOR CSP ([AFdeR80])

We first give an overview of some ideas of this proof system as used in the sequel. The following three facts concerning the syntax and semantics of CPS suffice for our purpose:

(1) The basic command of CSP is \([P_1 \parallel \ldots \parallel P_n]\) expressing concurrent execution of the (sequential) processes \(P_1, \ldots, P_n\).

(2) Every \(P_i\) refers to a statement \(S_i\) by \(P_i::S_i\). No \(S_i\) contains variables subject to change in \(S_j\) \((i \neq j)\).

(3) Communication between \(P_i\) and \(P_j\) \((i \neq j)\) is expressed by the receive and send primitives \(P_j?x\) and \(P_i!t\), respectively. Execution of \(P_j?x\) (in \(P_i::S_i\)) and \(P_i!t\) (in \(P_j::S_j\)) is synchronized and results in the assignment of the value of expression \(t\) to the variable \(x\) (such a pair \(P_i!t, P_j?x\), is called a syntactically matching communication pair).

In the proof system, the component processes are first proven correct in isolation, thus giving proof outlines in which each component statement is associated with appropriate pre- and postconditions. These pre- and postconditions satisfy Hoare’s assignment axiom and proof rules for sequential programming constructs. They also satisfy the following two axioms, which are introduced to separate this first sequential stage of the proof from the second global stage, required by the nonsequential nature of process communication:
This implies that the assumption on (i.e. the postconditions of) the communication actions in the component proofs must be justified by testing them for compatibility, in order for these outlines to be combined; this is the cooperation test. For this test a general invariant (GI) is needed which expresses globally which of these communications occur, i.e., which of the syntactically matching communication pairs match semantically. To be able to define GI, we allow programs to be extended with assignments to so-called auxiliary variables. These are variables whose values do not influence the flow of control within the original program and which are used to record information about the occurring communications.

As the variables appearing free in GI have to be updated at times, GI cannot be expected to hold throughout the program. However, GI only concerns the communication actions, so updating of the GI variables can be restricted to bracketed sections \( <S_1; \alpha; S_2> \) each of which is associated with a (unique) send or receive primitive \( \alpha \) (the statements \( S_1 \) and \( S_2 \) perform the updating of the GI variables). Consequently, an additional function of the cooperation test is to check whether GI is left invariant by the updating of its variables in each bracketed section. Basically, the test is the following:

Suppose we have proof outlines for the component processes of a CSP program. Then these proof outlines cooperate w.r.t. GI iff for any syntactically matching communication pair \( \alpha, \bar{\alpha} \) with associated bracketed sections and assertions \( \{p\} <S_1; \alpha; S_2\} \{q\}, \{\bar{p}\} <\bar{S}_1; \bar{\alpha}; \bar{S}_2\} \{\bar{q}\} \) in the respective proof outlines, the following condition holds:

\[
\{p \land \bar{p} \land GI\} S_1; x:=t; S_2; \bar{S}_2\{q \land \bar{q} \land GI\},
\]

where \( x:=t \) is the result of performing the communication.

The definition can be paraphrased informally as follows: if execution in process \( P_i \) respectively \( P_j \) arrives at the communication \( \alpha \), respectively, \( \bar{\alpha} \) (or rather, at the associated bracketed sections), as expressed by satisfaction of \( p \), respectively, \( \bar{p} \), and if additionally, these can occur simultaneously, as expressed by satisfaction of \( p \land \bar{p} \land GI \), then the processes \( P_i \) and \( P_j \) may communicate by executing \( \alpha \) and \( \bar{\alpha} \), and after this action (and after leaving the bracketed sections) both the assumptions \( q \) and \( \bar{q} \), and GI should hold. Thus, the assumptions made in the proof outlines concerning the value transfer are justified. For future reference, we mention here that, relative to such assumptions (and the initial state), one might say that a process can commit itself to the precondition of the next bracketed section, if any.

For examples which show the necessity of something like GI and a cooperation test for a Hoare style proof system for CSP (lest one wants to lose the idea of component proofs) the reader is referred to [AFdeR80]; these examples are fully applicable to the current model.

Now, the following parallel composition rule applies.
proof outlines for \{p_i \cdot P_i::S_i \cdot q_i \}_{i=1 \cdots n} cooperate w.r.t. GI

\{ \land P_i \land GI \parallel [P_1::S_1 \parallel \cdots \parallel P_n::S_n] \parallel q_i \land GI \}

From this rule it becomes obvious that, although captured in terms of Hoare-style axioms and rules, the proof system is not a proof system in the formal sense of the word: the parallel composition rule does not deal with Hoare-triples but with proofs (or proof outlines). Stated differently, the cooperation test operates on a fixed set of proofs, one for every process, and thus introduces a higher order concept.

In section 5, we shall show how the notion of cooperation (and interference freedom) can be embedded in a formal proof system. One of the essential steps for this is to incorporate the commitments and assumptions at the front and rear of bracketed sections explicitly in the statement-specification (see section 5.3). In the remainder of this paper, the analogues of the above ideas will be developed for communication through remote procedure calls, specifically for characterizing Ada's rendezvous mechanism and for communication by the entry call mechanism of Communicating Modules. In particular, as the reader will notice, the cooperation test and the bracketed sections reappear in the rendezvous and the entry call rules.

3. A PROOF SYSTEM FOR CONCURRENT Ada ([GdeR84])

3.1. The Ada rendezvous mechanism

The CSP communication mechanism combines synchronization of processes with the execution of an assignment. In a similar fashion, Ada combines synchronization with the execution of a procedure call.

The analogues in Ada, of the CSP send and receive primitives are the entry call, \texttt{call P_i.a(e#x)}, and the accept statement, \texttt{accept e(\overline{u}\#\overline{v}) do S end}. Again, there is a notion of syntactic matching: an accept for an entry \texttt{a} in process \texttt{P_i} matches with any call (outside \texttt{P_i}) for this entry in \texttt{P_i} (see the above call and accept). Execution of a matching call and accept is synchronized and results in executing the accept-body in which the formal parameters in \overline{u} have received the values of the expressions in \overline{e} and those in \overline{v} have received the values of the variables in \overline{x}. On completion of execution of the accept body, the actual variables in \overline{x} receive the values of the ones in \overline{v}. Hence, \overline{u} is a list of formal value parameters and \overline{v} is a list of formal value-result parameters. In particular, formal value parameters are never assigned to. Only after the second communication (of the values of \overline{v}) does the calling process proceed again.

Note that in Ada there is one-sided naming: an accept only specifies an entry procedure but not its communication partner. Also, there can be more than one accept for the same entry.

The rest of this section is based on a CSP-like language in which the CSP communication primitives are replaced by Ada-like call and accept statements.
3.2. A proof system for Ada-CF

Again the proof system contains rules for the sole purpose of separating reasoning about process interactions, i.e., about Ada-rendezvous', from verification of the (sequential) process bodies:

\[
\begin{align*}
\{ p \} \ \text{call} \ P_i.a(\vec{e}\#\vec{x}) \ \{ q \} \ & \text{provided} \ FV(p) \cap \{ \vec{x} \} = \emptyset \quad (1) \\
\{ p' \} \ S \ \{ q' \} \ & \text{provided} \ FV(p,q) \cap \{ u,v \} = \emptyset \\
\{ p \} \ \text{accept} \ a(\vec{u}\#\vec{v}) \ \text{do} \ S \ \text{end} \ \{ q \} \ & \text{provided} \ FV(p,q) \cap \{ u,v \} = \emptyset \quad (2)
\end{align*}
\]

The assertions \( q, p' \) and \( q \) in the above rules, again, embody assumptions about the communications. Note that (2) does not enforce any relationship between the pre and post assertion of an accept body and the pre and post assertions of the accept statement. This is consistent with our strategy because communications occur when "entering" or "leaving" an accept statement. Here, too, a cooperation test must check the assumptions as embodied in \( q, p' \) and \( q \). The condition on the pre assertion in the call axiom is vital because this assertion is needed to characterize the invariant part of the caller's state during a rendezvous. The requirements on the assertions in (2) reflects the fact that the formal parameters \( u,v \) are local w.r.t. the accept body and hence should not appear in assertions "outside" the body.

Similarly as in the CSP proof system, a general invariant \( GI \) is introduced to model synchronization globally, and also a set \( AV \) of auxiliary variables, in order to express \( GI : AV \) is a set of variables such that \( x \in AV \) if \( x \) appears only in assignments \( y := t \) where \( y \in AV \). (Notice that \( AV \) is defined relative to a particular program and that variables in \( AV \) do not influence the value of the "real" program variables during execution of the program).

With the introduction of bracketed sections we diverge from the development of the the previous section. The function of bracketed sections is to confine the updatings of \( GI \)'s variables. This is necessary because \( GI \) is needed to find the semantic matches, hence must be valid whenever a process has to perform a communication. In particular, \( GI \) must be valid inside the body of an accept, because it may contain other calls and accepts. This motivates the following definition.

**Definition 1:**

A **bracketed section** is a construct of the form

1. \(<S_1;\ \text{call} \ P_j.pr(\vec{f},\vec{x});S_2>\) or
2. \(\text{accept} \ pr(\vec{u}\#\vec{v}) \ \text{do} \ T_1;>T <;T_2 \ \text{endaccept}\)

such that \(S_1, S_2, T_1\) and \(T_2\) do not contain any call- or accept statements. The \(T_1\), respectively, \(T_2\) part of the accept body will also be referred to as the **prelude**, respectively **postlude** of the body.

Consider the following matching call (within its bracketed section) and accept, with associated assertions:

\[
\{ p_1 \}<S_1;|p_1|\ \text{call} \ P.a(\vec{e}\#\vec{x});|q_1>|S_2>\{ q_1 \} \text{ (in process P)}
\]

*For the cognoscenti we mention that the sets of formal parameters and (global) variables are assumed disjoint. Thus, we also do not need block rules.*
According to the semantics, a rendezvous is equivalent to the following statement sequence being executed:

\[
S_1; v := e, x; S_2; S_2; \]

Hence, cooperation for this pair at least means that

\[ \{ p_1 \wedge p_2 \wedge GI \} S_1; v := e, x; S_2 \{ p_1 \wedge p_2 \wedge GI \} \]

must hold. A second communication takes place, assumptions about which are embodied in \( q_2 \) and in \( \bar{q}_2 \) (or \( q_1 \)). Hence, the other requirement for cooperation is validity of

\[ \{ p \wedge q_2 \wedge GI \} S_2; x := v; S_2 \{ p \wedge q_2 \wedge GI \} \]

where \( p \) is an assertion that characterizes \( P \)'s state during the rendezvous. Because the state of \( p \) is not affected by executing \( S \), we may take \( p \equiv \bar{p}_1 \).

This gives the basic cooperation test. There is however a more elegant version possible. This other version capitalizes on the following well-known result about sequential procedure calls [A81]:

**Theorem 1:**

If in a call

1. the value-result parameters are pairwise disjoint,
2. no value-result parameter appears free in a value parameter and
3. no variable in the actual parameter list appears free in the procedure body

then, assignment of the actual to the formal parameters is equivalent to textual substitution of the actual for the formal parameters in the procedure body

The third restriction is automatically obeyed. *Now, let us impose the first two restrictions, too.* The following cooperation test ensues [GdeR84]:

**Definition 2:**

Proof outlines for processes \( P_1, \ldots, P_n \) cooperate w.r.t. \( GI \) if for any syntactically matching pair (with associated assertions)

\[
\{ p_1 \} S_1; S_2 \{ q_1 \} \quad \text{(in } P_i \text{)}
\]

\[
\{ p_2 \} \text{accept } a (\bar{e} \# x) \quad \text{(in } P_j \text{)}
\]

the following Hoare-triples can be proved

\[
\{ p_1 \wedge p_2 \wedge GI \} S_1; S_2 \{ p_1 \wedge p_2 \wedge GI \}
\]

\[
\{ \bar{p}_1 \wedge \bar{p}_2 \wedge GI \} S_2 \{ q_1 \wedge q_2 \wedge GI \}
\]

where \( [\bullet] \equiv [e, x, \bar{u}, \bar{v}] \).

Note that because syntactic substitution is used in this version, the condition in the call axiom, hence on \( \bar{p}_1 \), becomes essential.

With this version of the cooperation test we have obtained a verification method that is canonical in the following sense: When verifying a program we construct a proof (outline) in which with every syntactic statement, and in particular every call and accept, is associated one pre and one post assertion satisfying the cooperation test. This, notwithstanding the fact that every call has its own
characteristic synchronisation behaviour and its own characteristic parameters. In other words, the assertions in the proof are strong enough to justify the assumptions of each matching call and, symmetrically, are weak enough to remain valid under the "value injection" for each matching call. It is its canonical nature, that makes this verification method a usable one.

In [GdeR84] the Ada proof system is discussed in full detail, including

- A carefully selected series of "simplest" paradigms indicating why a global invariant and the definition of bracketed sections, as given above, are needed. These examples illustrate that without these concepts our proof system would be both incomplete and unsound.
- The extension of these techniques to prove general invariance or safety properties and in particular deadlock freedom and absence of failure.
- A proof of partial correctness, of termination, of absence of deadlock, and absence of failure, of a bounded buffer program.
- Correctness proofs of similar properties of a distributed priority queue.

4. OWICKI'S PROOF SYSTEM FOR SHARED VARIABLE CONCURRENCY ([OG76])

Owricki's proof system for concurrency with communication by means of shared variables is reviewed for her General Programming Language (GPL).

Concurrency is introduced in GPL by the cobegin statement:

\[
\text{cobegin } S_1 \parallel \cdots \parallel S_n \text{ coend.}
\]

Here, \( S_1 \) through \( S_n \) are sequential statements, called the component processes of this statement. A set of variables is shared between \( S_1, \ldots, S_n \), and the processes are executed in parallel. However, Owricki makes an essential restriction on the parallelism in GPL; this restriction is worked out in [deR85]. To simplify matters, this restriction is paraphrased as follows: the assignments and tests within the \( S_i \)'s are executed as atomic, indivisible actions. This means that the execution of a cobegin statement amounts to an arbitrary interleaving of these assignments and tests within the component processes.

In the GPL proof system, the components of a cobegin statement are proven correct in isolation, and, as in the CSP system of section 2, a consistency check is therefore needed when these separate proofs are combined. This check is in fact straightforward.

Consider the program cobegin \( x := 1; x := x + 1 \parallel x := 2 \text{ coend} \) and the (valid) isolated proof outlines \( \{\text{true}\} x := 1; \{x = 1\} x := x + 1 \{x = 2\} \) and \( \{\text{true}\} x := 2 \{x = 2\} \). A quick inspection of the program shows that after termination, the value of \( x \) is either 2 or 3. However, restricting one's attention to these component proofs in isolation might lead one to conclude that \( \{x = 2\} \) holds after termination. Now consider the precondition \( \{x = 1\} \) of the action \( x := x + 1 \), which is also the postcondition of \( x := 1 \). This assertion holds immediately after the assignment \( x := 1 \) has terminated, but this does not imply that \( \{x = 1\} \) holds whenever the second assignment, \( x := x + 1 \), is executed; The statement \( x := 2 \) could have been executed first. Hence, the necessary consistency check simply is the test whether
each assertion in the proof outline of a component process is invariant over all atomic actions of the other processes which can be interleaved at that point. This check implies that the postcondition of some action can safely be taken as the precondition of the following action in the component process. The following interference freedom test formalizes this idea:

Consider a program cobegin $S_1 \parallel \ldots \parallel S_n$ coend and proof outlines for the component processes. For each statement $T$, let $pre(T)$ respectively $post(T)$ denote the pre- respectively postcondition of $T$ within the proof outline. For any statement $A$ in $S_i$ and assignment $B$ in $S_j$, $i \neq j$, the following Hoare formulae should be valid:

1. $\{post(A) \land pre(B)\} B \{post(A)\}$
2. $\{pre(S_i) \land pre(B)\} B \{pre(S_i)\}$

The preconditions in this test may be interpreted as stating the possibility to be both at the beginning of $A$ and at the beginning of $B$, respectively, at the beginning of $S_i$ and at the beginning of $B$.

In Owicki's case this test is complicated by the presence of await-statements (which should not be confused with the wait-statement of the next section).

This simple expression of the interference freedom test changes drastically when considering Communicating Modules, next. There, the grain of interleaving, being an assignment in the present section and hence syntactically determinable, is determined semantically. Also the number of concurrent subprocesses varies dynamically in case of Communicating Modules, necessitating canonical interference freedom tests. Thus dynamic variation is again captured syntactically, as in the case of Ada's rendezvous rule.

As we observed about the the CSP proof system, the Owicki system too, is not a proofsystem in the formal sense of the word; and for the same reason. Section 5 shows how to incorporate interference freedom in a formal way.

5. A SYNTAX DIRECTED PROOF SYSTEM FOR COMMUNICATING MODULES

The purpose of this section is twofold: Firstly, to show how the notion of cooperation can be embedded in a formal syntax-directed proof system. Secondly, to solve the verification problem for monitor-like languages. This chapter is based on initial work as reported in [RDKdeR81, GdeRR82a, GdeRR82b].

Monitors combine (1) the notion of abstract data type with (2) a notion of concurrency that derives from the logical independence of the various data types in a system. Interaction within a system occurs through the data type interfaces. The logical integrity of the data type abstraction and the functionality of the operations are maintained (1) by imposing strict synchronization between a monitor and the entity requesting its service, while performing this service and (2) by allowing a monitor to be serving at most one request at a time. This last fact does not imply that the data type operations of a monitor are serialized, but only that the (partial) execution of separate data type operations have to be strictly interleaved.

On a more concrete level, when a monitor is servicing a requestor, hence is performing a data type operation, the requestor is said to be active in the monitor.
and to have acquired the monitor-lock, effectively locking out the servicing of any other requestor until the former relinquishes the lock. At such a time, a monitor may start (again) servicing someone else.

The next section describes the language CM. From this, it will become clear that modules make concrete the two essential features of monitors: a procedural (date type) interface with the environment and a way to switch activity within a monitor, between different requestors.

5.1. The language Communicating Modules (CM)

Modules are a generalization of the usual monitor. Originally, a monitor was a passive object only reacting to outside requests. Modules, while able to react to requests, are active entities and may also initiate communication actions. As such, a module combines, e.g., the notions of monitor and process from Concurrent Pascal ([BH75]) and is akin to a DP-process ([BH78]).

A CM program consists of a fixed set of concurrently executing modules: \( [M_1, \ldots, M_n] \). Modules do not contain other modules or CM programs, nor do they share variables. The syntax of a module is:

\[
M_i :: <\text{entry procedure}\> s : [ <\text{(initial) statement}> ]
\]

A module \( M \) is executed by starting execution of its initial statement; this may then be interleaved with execution of calls of other modules to the entry procedures of \( M \). From the caller's viewpoint, execution of an entry call is identical to an Ada rendezvous. Accepting a call represents the only form of communication between modules, and results in a new procedure instance being created and executed. Similar to Ada's entry call, during execution of a CM call, the caller is suspended until the callee has finished execution of the corresponding instance.

The syntax of an entry procedure is:

\[
<\text{name}> (<\text{input parameter list}> \ # <\text{output parameter list}>): [ <\text{statement}> ]
\]

The parameter mechanism is call-by-value-result; within a procedure body no assignments to input parameters are allowed. Parameters are local w.r.t. the procedure body. Procedure bodies and initial statements are ordinary sequential statements, constructed using sequential composition, if, and while-statements. The primitive actions are the assignment, external request or call statement (see section 3) and the wait-statement, denoted by \text{wait}.

Waits are the primary way to establish synchronization (to be defined below).

Synchronization occurs in a module:

- when arriving at a wait-statement.
- upon completion of execution of its initial statement,
- upon completion of a call to one of its entry procedures.

Such points are called \text{waiting points}.

By synchronization we mean either:

- the act of honouring an arbitrarily selected entry call, or
the act of passing some wait-statement, having been a waiting point earlier during execution and, hence, possibly different from the one execution just arrived at. Observe that on arrival at a waiting point, the monitor lock is released.

Note that the more general construct, await \(b\) of GPL, which suspends execution until after the boolean expression \(b\) becomes true, can be simulated by

\[
\text{while } b \text{ do wait od.}
\]

The main characteristic of CM, on which the proof system is built, is the *synchronization-at-waiting-points*. A clear picture of this is essential for understanding the proof system. Figure 1 below shows the execution of 3 modules, \(M_1\), \(M_2\) and \(M_3\). The horizontal lines represent the time axes for each of the respective modules, along which execution proceeds. During execution of \(M_1\) an external request is made, the action of which is atomic w.r.t. \(M_1\), as indicated by the brackets "\(\langle\)" and "\(\rangle\)". Execution of \(M_2\) proceeds until a waiting point is encountered denoted by the (leftmost) vertical wriggles, at which synchronization takes place. \(M_2\) continues by honouring \(M_1\)'s request and starting execution of a new instance of \(a_2\). During this execution, other waiting points are encountered and external requests are made. Finally execution arrives at the end of procedure body, after which \(M_1\) and \(M_2\) proceed independently. At parameter passing, external information from the caller (\(M_1\)) will be injected into the callee (\(M_2\)) by the input parameters at the beginning of the procedure execution and vice versa at the end.

The *internal* mode of execution of each CM module can be viewed as the concurrent execution of a set of subprocesses - its initial statement and its procedure instances - sharing the local variables of that CM module. This execution is *concurrent* in the sense of section 4: A possible execution of these subprocesses is an...
arbitrary nondeterministic interleaving of the "atomic" actions of these processes.

One should keep the following essential differences with section 4 in mind:

- an atomic action now extends from one waiting point to another, and hence is not anymore a syntactically determined action, and
- the number of subprocesses, active at a certain time, is dynamically varying as it comprises all instances of the entry procedures.

5.2. Program specifications and problems

This section addresses the problem of constructing a formal syntax-directed proof system. Initially, this is a question of finding the right kind of program specification. As a consequence, we need to address some of the verificational problems that CM poses.

We want the verification method to be casted in terms of formal rules and axioms and to be syntax-directed. In other words, we want a set of rules and axioms, all "working" on the same type of formulae or specifications. Moreover, the resulting system should support proofs of programs that follow the syntactic structure of these programs.

5.2.1. Communication

Consider the CSP and Ada proof systems of section 2 and 3. Such systems do not pass our requirements because the parallel composition rules (through the cooperation test) refer to the actual proofs of the component processes. This is necessary as the various communication assumptions, as expressed by the post assumptions of bracketed sections, have to be validated and such assumptions are buried within the component proofs.

Obviously, to construct rules of the required form, such assumptions will have to be made explicit. Thus, specifications have a form like \(<AS \| p \| S \{ q \} \rangle\) where \(AS\) embodies in some form the communication assumptions that have been made in order to derive \(\{ p \| S \{ q \} \}\). The parallel composition rule discharges these assumptions by requiring that cooperation be proved. The specific form of \(AS\) (in case of CM) is discussed in section 5.4.

Two facts complicate this picture. The first is caused by the cooperation test itself; the second one by the semantics of CM.

(1) Cooperation means that two communicating modules, together transform their states, satisfying two pre assertions, into states satisfying two post assertions (that embody communication assumptions). This dependence on the assertions describing the states in which the various modules enter communication must be made explicit, too. Obviously, these assertions are the pre assertions of the bracketed sections, and will be considered as commitments.

(2) A CM module may engage in communication, i.e., may honour an entry call, at any waiting point. Question: How do we characterize the state in which an
entry procedure starts executing? This is essential, both for verifying the entry procedure and for the cooperation test. The answer is as old as it is obvious [Hoa74]: Introduce a module invariant, $MI$, which characterizes the state of a module at any waiting point.

At this point, a specification looks like $<AS, CO, MI \mid \{p\}S\{q\}>$, where $CO$ (commitment) describes the states in which $S$ enters into communication. Its meaning is:

During any execution of $S$ which starts in a state satisfying $p$, 
1) $MI$ will hold at waiting points,
2) $CO$ will hold in those states in which $S$ performs communications, and
3) $q$ will hold if and when $S$ terminates,

**PROVIDED**
every state produced by a communication involving $S$ satisfies $AS$.

### 5.2.2. Internal execution

As stated before, internal execution of a CM module amounts to an interleaving of actions from the initial statement and the procedure instances, which are bounded by waiting points.

On the level of verification this mode of execution is to be captured - as in section 4 - by interference freedom conditions. The interference freedom test basically prescribes that assertions associated with the interleaving points of one process, be left invariant by the interleavable actions of the other processes. There is a different way of expressing this:

Take any assertion, $p$, associated with an interleaving point in a process. If execution in any other process starts or resumes in a state in which $p$ holds and if execution reaches another interleaving point then $p$ should hold again.

In this formulation, a more operational view is taken of the issue. A view that is also the more basic one, since it applies equally well to our CM as it does to Owicki's GPL. It allows for a concise explanation of interference freedom of an assertion, $p$, with respect to all the atomically executed actions of another process, as will be explained next.

Consider the intermediate assertions generated in a proof of, say, $\{p\}S\{q\}$. Better yet, consider it as a proof outline in which these assertions appear explicitly associated with the front and rear of the statements contained in $S$. In this form it is particularly obvious that such a proof outline is a syntactically sugared Floyd-style inductive assertion proof. This association makes it clear that validity of a proof (outline) for $\{p\}S\{q\}$ implies that whenever execution starts before or after a statement of $S$ in a state that satisfies the associated assertion, then, if the front or rear of any statement is reached it will be in a state that satisfies the corresponding assertion. (This paraphrases Floyd's definition of local correctness of a labeled program scheme [Liv78].)

If this statement is compared with the operational characterization of interference freedom, the following idea for proving interference freedom of an assertion $p$
w.r.t. a CM-statement \( S \) (with pre assertion \( p \)) ensues:

Construct a proof for \( \{ p \wedge \neg f \} S \{ p \} \) in such a way that

1. the pre assertion of any wait in \( S \) implies \( p \)
2. the post assertion of any wait in \( S \) implies \( p \)

Condition (1) ensures that \( p \) holds as commitment at any (internal) waiting point. Condition (2) means that execution at a waiting point resumes, by assumption, with \( p \) true again. This is necessary because if execution resumes after a wait, some unknown action(s) have been interleaved which may have invalidated \( p \). As we want only to show that the actions of \( S \) do not interfere, we have to make this assumption.

To formulate this interference freedom test correctly, also the pre and post assertions of waits have to be made explicit. Thus, we arrive at the final form of CM specifications:

\[
<CA, WA, CO, MI | \{ p \} S \{ q \}>
\]

Here, \( CA \) collects the communication assumptions and \( WA \) the waiting assumptions about the states in which execution of \( S \) resumes at waiting points. Commitment \( CO \) describes the states in which 1) \( S \) enters communication and 2) \( S \) arrives at a labeled waiting point. \( MI \) is the module invariant. The precise meaning of these specifications is given and discussed in section 5.4.1.

This characterization of control flow is illustrated below.

Consider \([M_1 | M_2 | M_3]\), declared as in figure 2: Execution of \([M_1 | M_2 | M_3]\) can be graphically represented as in figure 3.

First the initial statements of \( M_1, M_2, M_3 \) are executed, leading to the call \( M_2.a_2(x#y) \) in \( M_1 \) and waiting point (3) in \( M_2 \). Then \( a_2 \) is executed, leading to waiting point (1); \( c \) being true, at (3) execution is resumed, leading to the call \( M_3.a_3 \) and waiting point (4); upon which, \( c \) being false, execution of \( M_2.a_2 \) resumes at (1) and finishes at (2). Consequently in \( M_2, MI_2 \), characterizing the initial state for calls to \( M_2 \), should be first established at (3), then at (1), then at (4), and finally at (2). In the figures, the leftmost (3), (1) and (4) label the events of arriving at waiting points, and should have assertions \( CO_2 \) and \( MI \).
attached. The rightmost (3) and (1) label resumption events and should have assertions \(WA_2\) and MI attached.

This is illustrated in figure 4. There, the previous figure is overlayed with these assertions. Also, the assertions that should hold before \((CO_2, GI)\) and after \((CA_2, GI)\) bracketed sections have been indicated.

5.3. The proof system

5.3.1. The formalism

In the previous section, the general form of specifications has been introduced. The formal details must be filled in now. In particular, we must decide on the form of the various assumptions and commitments.

These assertions are attached to various points in the program: Waiting assumptions are associated with waiting points (when execution resumes); communication assumptions with the closing brackets of bracketed sections; commitments with the opening brackets and, again, with waiting points (when arriving at such a point). To make these associations, labels are used. Before defining these, bracketed sections need to be introduced.

**Definition 3:**

- A module is *fully bracketed* iff
  
  1. every call, \(C\), appears within a unique *bracketed section* \(<S_1; C; S_2>\),
  
  2. every procedure \(pr(u#v); [S]\) is uniquely bracketed in the sense that
S = S_1 > S_2 < S_3 (S_1 is called pr's prelude; S_2 its postlude)

(3) no bracket is introduced otherwise
(4) the statements S_1 and S_2 in both (1) and (2) do not contain any calls or waits.

• A program is **fully bracketed** if all of its constituent modules are.

**Definition 4:**
• A module is **fully labeled** if:
  (1) every opening bracket (<) and closing bracket (>) has a label: l.< respectively>
  (2) every wait is labeled at its front and rear: l.wait.l'
  (3) every call is labeled at its front: l.call Pi.prj(x#x)
  (4) the rear of the initial statement is labeled: S.l
  (5) the front of every procedure body and the initial statement is labeled: l.S .
  (6) no label is introduced otherwise
  (7) all labels are unique
• A program is **fully labeled** if each of its constituent modules is.

Clauses (3), (4) and (5) need clarification. Clause (3) is forced by the cooperation test, structured as in section 3, which needs a characterization of the module state during a call. The label of a call is interpreted as labeling this point. When every module in a program has completed executing its initial statement, the program terminates. It follows that the state in which an initial statement terminates is of significance. This point, however is also a waiting point, which im-
plies that the post assertion of the initial statement must be interference free. Hence clause (4). Clause (5) is needed in order to properly express the interference freedom conditions; see section 5.3.3.

Definition 5:

(1) $CA, WA$ and $CO$ are sets of labeled formulae: $\{ l: p | l \text{ a label}, p \text{ an ordinary first order assertion} \}$.

(2) Let $S$ be a set as in (1) and $l$ a label. Then $S(l) = \bigwedge \{ p | l: p \in S \}$.

Note that by definition, $S(l) \equiv \text{true}$ if there is no $p$ such that $l: p \in S$.

(3) A set $S$ as in (1) is disjoint if there is no label, $l$, and assertions, $p, q$, such that $l: p, l: q \in S$ but $p \not\equiv q$.

This is a straightforward definition. Predictably, the module-state will now include truth-values for labels, too, in order to express which labeled points have been reached. If execution arrives at a labeled point (labeled, say, with $l$), and only then, the corresponding state will associate the value true with this label. A commitment or assumption $A$ is satisfied in a state, precisely if for any label $l$ which is true in this state, $A(l)$ is true in this state.

A set of labeled assertions is disjoint, precisely if with every label there is associated a unique assertion. This notion is needed in the formulation of the proof rules in order to express that specifications of two statements can only be combined if they do not specify the states at each other's labeled points.

Definition 6:

Specifications are of one of the following forms:

(1) $<CA, WA, CO, MI | \{ p \} S \{ q \} >$, $S$ some statement or procedure.

(2) $<CA, \emptyset, WA, MI | \{ p \} M \{ q \} >$, where $M$ is a module.

(3) $<\emptyset, \emptyset, \emptyset, \text{true} | \{ p \} P \{ q \} >$, where $P$ is a program,

provided $CA, WA$ and $CO$, part of the specification of some $T$, obey the following conditions:

(1) any label in $CA$ labels a closing bracket in $T$,

(2) any label in $WA$ either labels the rear of a wait in $T$ or $T \equiv \text{true} | T' . l$ or $T \equiv T' . l$,

(3) any label in $CO$ labels an opening bracket, a call or the front of a wait in $T$,

(4) $CA, WA$ and $CO$ are disjoint,

(5) $MI$ does not contain formal parameters and in any $l: p$-term, $p$ only may contain formal parameters of the (possible) procedure in which $l$ appears,

(6) the pre and post assertions in a procedure specification may not contain any formal parameter.

Statements and procedure specifications are relative with respect to both communication and waiting point assumptions. This stands to reason since neither the calling environment nor the interleavable actions are known. Given the module text containing the statement or procedure, the latter are known and hence a module's specification does not depend on waiting point assumptions anymore. Likewise, the program text determines the calling environment of a module and hence program specifications are not relative to any assumption.
Observe that a program specification has trivial CO and MI components and hence is a pure partial correctness specification. The reason is not that the proof system would not support proofs of invariance properties - it would -, but rather that it is awkward to define validity of such specifications in a cooperation-based framework; see [GdeR84] for an attempt.

Definition 7:
\(<CA, WA, CO, MI \mid \{p\}S\{q\}>\) is valid iff for any (partial) execution of \(S\) that starts in a state satisfying \(p\) and for which \(CA\) has been continuously true, if either from the start or from some waiting point onwards \(WA\) starts to hold and continues to hold, then \(CO\) will hold in the last state of the execution sequence, \(MI\) will hold if this is a waiting point and \(q\) will hold if \(S\) has terminated.

5.3.2. A sampler of proof rules

Only the interesting rules are discussed here. I.e., only the rules that involve statements causing synchronization to occur. The rest of the proof system is listed in section 5.3.5.

5.3.2.1. wait-rule

\[ p \rightarrow q \land MI \]
\[ <\emptyset, \{l':r\}, \{lq\}, MI \mid \{p\}l.wait(l|r)> \]

This statement signals a waiting point. Hence \(MI\) and \(CO(l)\) should hold, which is enforced by the premiss. On resumption, the strongest claim that can be made, is that \(WA(l')\) holds.

5.3.2.2. call-rule

\[ <\emptyset, \emptyset, \emptyset, \text{true} \mid \{p\}S_1\{q\}> , <\emptyset, \emptyset, \emptyset, \text{true} \mid \{q\}S_2\{r\}> \]
\[ <\{l':s\}, \emptyset, \{l:p, l:q\}, \text{true} \mid \{p\}l.\<S_1;\text{call} M_1.a_j(e)x;S_2;\text{call} M_2.a_k(e)x\}> \{r \land s\} > \]

This is a complicated rule. First observe that the rule applies to a call within its bracketed section. Within our framework, this makes sense. Disregarding synchronization (which we consistently do at this level) a call acts as a random assignment to its value-result parameters, as far as the calling module is concerned. A random assignment, \(x := ?\), is axiomatized, in analogy with an ordinary assignment, by \(\{\forall x \mid x := ? \{p\}\} \) (cfr. [APB7]). This explains the relationship between the post assertion of the first premiss and the pre assertion of the second. The post assertion of this last premiss, characterizes the terminal state of the bracketed section in so far as it is independent from the values assigned to the value-result parameters. For the parameter-dependent part, we cannot do better than to resort to \(CA(l')\); an assumption that will be validated in the cooperation test. And this gives the post assertion of the conclusion of the rule.

5.3.2.3. procedure-rule

\[ <\emptyset, \emptyset, \emptyset, \text{true} \mid \{p \land \bar{p}\}S_1\{q\}> , <CA, WA, CO, MI \mid \{q \land r\}S\{s\}> \]
\[ <\emptyset, \emptyset, \emptyset, \text{true} \mid \{s\}S_2\{t\}> , s \rightarrow s , t \rightarrow MI \]
\[ <CA \cup \{l:r\}, WA \cup \{l:\bar{p}\}, CO \cup \{l':s\}, MI \mid \{p\}a(e\#x)\{l.S_1;\text{call} M_1.a_j(e)x;S_2;\text{call} M_2.a_k(e)x\}> \{l.S_1;\text{call} M_1.a_j(e)x;S_2;\text{call} M_2.a_k(e)x\}> \]

provided $FV(p, x) \cap \{\bar{u}, \bar{v}\} = \emptyset$ and $CA \cup \{l: r\}$, $WA \cup \{l: p\}$ and $CO \cup \{l': s\}$ are disjoint.

The same remarks as for the call-rule apply here. The rear of a procedure body is a waiting point, so that $MI$ must hold there. It is, however, a waiting point where execution does not resume again. So, no waiting assumption or commitment is associated with it.

5.3.2.4. module-rule

Let module $M_i$ be defined as

$$M_i :: a_1(\bar{u}_1 \# \bar{v}_1); [l_1 \cdot S_1]$$

$$\ldots$$

$$a_k(\bar{v}_k \# \bar{v}_k); [l_k \cdot S_k]$$

$$: [l_0 \cdot S_0 \cdot l_0']$$

$$<CA_j, WA_j, CO_j, MI \mid \{MI\} a_j(\bar{u}_j \# \bar{v}_j); [l_j \cdot S_j] >, MI \rightarrow WA_j(l_j) \quad j = 1 \ldots k$$

$$<\bigcup_{i=0}^k \overline{WA}_i \#_{i,j} (CA_j, WA_j, MI, \{\bar{u}_j, \bar{v}_j\}, a_j(\bar{u}_j \# \bar{v}_j); [l_j \cdot S_j]) >, j = 1 \ldots k$$

$$\bigcup_{i=1}^k \overline{WA}_i \#_{i,0} (CA_0, WA_0, p, \emptyset, l_0, S_0)$$

for every $l.wait.l'$ in $M_i$: $\bigcup_{i=0}^k CO_i(l) \rightarrow \bigcup_{i=0}^k WA_i(l') \land MI$

provided

- $WA_0 \equiv WA_0(l_0) \cup \{l_0: WA_0(l_0)\}$,
- $\overline{WA}_i \equiv WA_i(l_i: WA_i(l_i)) \quad i = 0..k$ and
- $\bigcup_{i=0}^k CA_i$, $\bigcup_{i=0}^k WA_i$ and $\bigcup_{i=0}^k CO_i$ are disjoint.

The first $k+1$ specifications in the premisses enforce (sequential) correctness of the entry procedures and the initial statement. The premisses on the third and fourth line are the more interesting ones, dealing with interference freedom. They denote sets of interference freedom specifications. Their discussion, however, is postponed to section 5.4.3. Here, we only mention that in the right-hand-side argument of $\#_{i,k}$, the last entry is the statement, $S$, whose actions must not interfere with $\overline{WA}_i$. The first two entries are the two assumptions used to prove $S$ sequentially correct. The next entry characterizes the state in which $S$ starts executing. Finally, we need to know the local variables of $S$.

The first-order formulae in the first two lines force the waiting assumptions associated with the front labels of $M_i$'s procedures and initial statement to be trivial in the sense that they are implied by the assertions ($MI$ or $p$) characterizing the state in which execution starts at these points. Since $MI$ and $p$ are interference free by definition, this justifies the use of $\overline{WA}_i$ instead of $WA_i$ in the in-
terference freedom expressions. Finally, the premisses on line 5 imply that also in
the case that execution decides just to pass a wait without interleaving any ac-
tions, the waiting point assumption can be assumed to hold. Note that the conclu-
sion of the rule does not make waiting assumptions anymore.

This is a complicated rule with many premisses. However, the thing to
remember is that none of these premisses depend on proof outlines anymore.
Specifically, in section 5.3.3, the interference freedom specifications will be defined
in terms of the specifications of the initial statement and the entry procedures.

5.3.2.5. program-rule

\[
\begin{align*}
&\text{program-rule} \\
&\quad <CA_i, \emptyset, CO_i, M_i | \{p_i\}M_i | q_i > \quad i = 1...n \\
&\quad \text{Coop} ( \{<CA_i, \emptyset, CO_i, M_i | \{p_i\}M_i | q_i > \} | i = 1...n ), GI \\
&\quad <\emptyset, \emptyset, \emptyset, true | \{p_1 \land \cdots \land p_n \land GI \} | [M_1 | \cdots | M_n ] | q_1 \land \cdots \land q_n \land GI >
\end{align*}
\]

provided

- \(k \bigcup_{i=1} CA_i \), \(k \bigcup_{i=0} WA_i \) and \(k \bigcup_{i=0} CO_i \) are disjoint,
- \(FV(CA_i, CO_i, M_i, p_i, q_i) \) does not contain any variable of \(M_j \) (\(i \neq j \)),
- no variable free in \(GI \) appears on the left-hand-side of any assignment out-
side a bracketed section, nor as parameter of any call or procedure.

The premiss in the second line denotes a set of cooperation specifications.
Again the precise definition of cooperation is postponed (to section 5.3.4). Its
definition is analogous to the Ada-test, but for a slight complication caused by the
interference freedom test.

The conclusion of the rule does not depend on any assumption anymore.
Also, the commitments and module invariants have been trivialized, for the rea-
sons stated in 5.3.1.

Here, too, we stress that the cooperation specifications are defined in terms of
the module specifications only.

5.3.3. Interference freedom

Fix an assertion \(p \) and a statement \(S \). \(S \) has a (sequential) proof
\(<CA, WA, CO, MI | \{F\}S | q > \). We want to show that \(S \) is interference free
w.r.t. \(p \). The conceptual discussion of section 5.2.2 is the starting point.

Using the notation that has been developed above, the suggested characteriza-
tion of interference freedom of \(p \) w.r.t. \(S \) can be expressed as the validity of the specification

\[<CA, WA', CO', MI | \{F\}S | p > \], where

\(WA' \) and \(CO' \) are defined in such a way that for any label \(l \), if \(l \) labels the front of
a wait in \(S \) or \(S \) itself then \(WA'(l) \equiv WA(l) \lor p \) and if \(l \) labels the rear of a wait
in \(S \) then \(CO'(l) \equiv p \).

We translated the requirements on the pre and post assertions of waits into
changes of \(WA \) and \(CO \).
This seems straightforward, but we should look at it more closely. The first complication arises as follows: the assertion \( p \) is a waiting point assumption and \( S \) is the body of a procedure or an initial statement. A possibility is that \( p \equiv WA(l) \) for some \( l \) in \( S \) itself: In case \( S \) is a procedure body for which at least two instances coexist, \( WA(l) \) must be interference free w.r.t. \( S \). The formal parameters of a procedure are local w.r.t. its body. Such parameters will occur both in \( S \) and in \( WA(l) \). However, these occurrences belong to different instances.

Conclusion: formal variables common to both \( WA(l) \) and \( S \) must be renamed.

As to the second complication, consider the program \( [M_1 \parallel M_2] \) declared by

\[
\begin{align*}
M_1 :: & a_1(\#): [S] \\
& \vdots \\
M_2 :: & \text{[call } M_1.a_1(\#).] \\
\end{align*}
\]

where \( S \equiv S' \land \text{wait } l'; S'' \), and the wait statement shown is the only one in \( M_1 \).

Let \( p \equiv WA_1(l') \). Clearly, \( p \) need not be proved invariant over \( S \) since there can be no coexisting instances of \( a_1 \). In other words,

\[
<CA, WA', CO' , MI \ 1 [MI \land p ] S \{p \}>
\]

should hold trivially. But, does it?

This specification holds trivially, only if \( MI \), \( p \) and \( WA_1(l') \) are such that both \( MI \land p \) and \( WA_1(l') \land p \) are equivalent to false, since only in this case none of the commitments \( (CO, MI \) and \( q) \) need be established. Now, observe that \( MI \) must hold at any waiting point and that \( p \equiv WA_1(l') \). So, the two conjunctions above, will only yield \( \text{false} \) if \( WA_1(l') \equiv \text{false} \). Clearly, this is nonsensical since it would imply that execution will never be at the point labeled \( l' \), while we merely want to express that execution is never going to be at this point simultaneously in two different instances of procedure \( a_1 \).

Conclusion: We cannot distinguish between one instance of \( a_1 \) being able to resume at \( l' \) and two instances of \( a_1 \) being able to resume at \( l' \).

The solution is to introduce so-called instance counters.

- for each procedure \( a_j \) in module \( M_i \), a (proof) variable \( i_{c_{ij}} \) is introduced, which counts the number of coexisting instances of \( a_j \),
- \( i_{c_{ij}} \) is a reserved name, not occurring in the program text,
- \( i_{c_{ij}} \) may only appear free in GI and in assertions associated with labels appearing in \( a_j \).

The interference freedom test will be modified in such a way that when it operates on assertions and statements from the same procedure (implying the existence of multiple instances), it will, e.g., allow in the above case the use of an extra assumption, \( i_{c_{11}} \geq 2 \), when execution starts or resumes at a waiting point.

The assertions \( WA(l') \) and \( p \), belonging to the specification of \( a_1 \), will contain terms \( i_{c_{11}} \leq 1 \). Clearly, the interference freedom specification of the above example will hold trivially now. Note, however, that we only managed to exchange one problem for another: now, we need to check whether assumptions like \( i_{c_{11}} \leq 1 \) are
in fact true. Such assumptions can be resolved during the cooperation test only, and hence its discussion belongs to the next section.

We can now formally define the set of *interference freedom specifications*, denoted by the interference freedom expression in the module rule.

**Definition 8:**
Let $A \subseteq \{l:p \mid l \text{ a label}, p \text{ a first order formula}\}$.
Then the interference freedom expression
$$A#_{i,j}(CA, WA, MI, p, \bar{x}, S)$$
stands for the set of interference freedom specifications
$$\{<CA, WA(q' \land IC), CO, \text{true} \mid \{p\}S[q'] > 1 \mid l.q \in A\}$$,
where
$$q' \equiv q[\bar{y}/\bar{x}] \text{ (where } \bar{y} \text{ do not appear free in } CA, WA, CO, p, q \text{ or } S)$$
$$IC \equiv \begin{cases} ic_{ij} \geq 2 & \text{if } j > 0 \text{ and } l \text{ occurs in } S \\ \text{true} & \text{otherwise} \end{cases}$$
$$WA(r) \equiv \{l:WA(l) \land r \mid l \text{ labels the rear of a wait in } S, S \equiv lS' \text{ or } S \equiv S'.l\}$$
$$CO \equiv \{l:q' \mid l \text{ labels the front of a wait in } S\}$$

The cooperation test validates the communication assumptions, $CA$. Earlier, we stated that assumptions about the number of coexisting instances will be discharged during the cooperation test. Consequently, this will only happen if they become part of the communication assumptions, $CA$, of the *sequential* specification of the procedure in question. This will in fact be the case in any procedure specification that is part of a *provable* module specification, as is argued now.

In a provable procedure specification, such assumptions can only be introduced by a communication or waiting point assumption, since these are the only assertions that need not be derived in the proof of the specification. Suppose an assumption $ic_{ij} \leq 1$ is introduced via a waiting point assumption associated with a statement $l.wait.l'$. Then also $CO(l) \land MI \rightarrow ic_{ij} \leq 1$, since the specification is to be part of a provable module specification and one of the module rule's premises forces that $CO(l) \rightarrow WA(l') \land MI$. But $CO(l)$ is a formula that must be derived during the proof of the procedure specification. Now, the only assumptions that are not constrained by the application of the module rule are the communication assumptions and hence $ic_{ij} \leq 1$ must be part of $CA$.

Section 5.4 contains an example of a proof.

5.3.4. The cooperation test

**Definition 9:**
The expression
$$\text{Coop}(\{<CA_i, WA_i, CO_i, MI \mid \{p_i\}M_i[\{q_i\}] > 1 \mid i = 1 \cdots n\}, GI)$$
stands for the set of *cooperation specification* pairs.
\[ \langle \emptyset, \emptyset, \emptyset, \text{true} \mid \{CO_i(t) \land M_j \land GI \} S_1; iC_{jk} := iC_{jk} + 1; S_1[[\bullet]] \{CA_j(t)[[\bullet]] \land GI \} \rangle, \]
\[ \langle \emptyset, \emptyset, \emptyset, \text{true} \mid \{\forall \vec{x} \ CO_i(\vec{t}) \land CO_j(\vec{t})[[\bullet]] \land GI \} S_2[[\bullet]]; iC_{jk} := iC_{jk} - 1; S_2 \{CA_i(\vec{t}) \land GI \} \rangle \]

where \([\bullet] = [\varepsilon, x, \emptyset / \vec{y}, \vec{v}]\), for any matching call-procedure pair

\[ l. \langle S_1 ; \llcorner \text{call } M_j, a_k (\vec{e} \# \vec{v}) ; S_2 \lrcorner; \rangle \quad \text{(in } M_i \text{)} \]
\[ a_k (\vec{u} \# \vec{v}); \llcorner S_1 [j'] ; \lrcorner; S_2; \llcorner \llcorner \llcorner \text{ (in } M_j \text{)} \]

Predictably, this test is based on the Ada cooperation test. There are some trivial cosmetic changes as the current specifications explicitly mention the communication assumptions. More interesting is the presence of the instance counters. Entering and leaving a procedure \((a_j \text{ in } M_i)\) has the effect of increasing and decreasing \(iC_{ij}\) by 1. These implicit assignments have simply been made explicit here.

5.3.5. A list of rules and axioms

To be complete, the syntactic constraints on CM programs that have been made implicitly, are listed here.

- The free variables of different modules must be disjoint.
- The same goes for the formal parameters. Moreover, the sets of formal variables and "ordinary" variables are disjoint.
- Module names are of the form \(M_j; j = 1, 2, \ldots \); procedure names have the form \(a_k; k = 1, 2, \ldots \).
- There is a set of variables \(\{iC_{ij} \mid i, j = 1, 2, \ldots \}\), disjoint from both the formal variables and the ordinary variables, which may not appear in the program text.
- Any label within a CM program may appear at most once.
- Variables \(iC_{ij}\) may only appear free in \(GI\) and in assertions associated with the body of \(a_j\) in \(M_i\).

Some notation: write \(\{p \mid S \{q\} \} \) instead of \(\langle \emptyset, \emptyset, \text{true} \mid \{p \} S \{q\} \rangle\).

Next, we turn to the proof rules and axioms.

5.3.5.1. assignment

\[ \{p [e/x] \} x := e \{p\} \]

5.3.5.2. wait see 5.3.2.1

5.3.5.3. call see 5.3.2.2
5.3.5.4. sequential composition

\[<CA_i, WA_i, CO_i, MI_i | \{p_i | S_i | q_i \}> \quad i=1,2, \quad q_1 \rightarrow p_2\]

\[<CA_1 \cup CA_2, WA_1 \cup WA_2, CO_1 \cup CO_2, MI_1 \vee MI_2 | \{p_1 | S_1 | S_2 | q_2 \}>,\]

provided \(CA_1 \cup CA_2, WA_1 \cup WA_2\) and \(CO_1 \cup CO_2\) are disjoint.

5.3.5.5. if

\[<CA_1, WA_1, CO_1, MI_1 | \{p \wedge b | S_1 | q \}>, \quad <CA_2, WA_2, CO_2, MI_2 | \{p \wedge \neg b | S_2 | q \}>\]

\[<CA_1 \cup CA_2, WA_1 \cup WA_2, CO_1 \cup CO_2, MI_1 \vee MI_2 | \{p \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid S_1 | q \}>,\]

provided \(CA_1 \cup CA_2, WA_1 \cup WA_2\) and \(CO_1 \cup CO_2\) are disjoint.

5.3.5.6. while

\[<CA, WA, CO, MI | \{p \wedge b | S | p \}>\]

\[<CA, WA, CO, MI | \{p \mid \text{while } b \text{ do } S \mid \text{ od } \{p \wedge \neg b \}>\]

5.3.5.7. initial statement

\[<CA, WA, CO, MI | \{p \wedge q | S | MI \}>\]

\[<CA, WA \cup \{l:q, l':r \}, CO, MI | \{p | \{l, S, l' \} | r \}>\]

provided \(WA \cup \{l:q, l':r \}\) is disjoint.

5.3.5.8. procedure see 5.3.2.3

5.3.5.9. module see 5.3.2.4

5.3.5.10. program see 5.3.2.5

5.3.5.11. consequence

\[<CA, WA, CO, MI | \{p | S | q \}>\]

for any label \(l: CA'(l) \rightarrow CA(l), WA'(l) \rightarrow WA(l), CO(l) \rightarrow CO'(l)\)

\[MI \rightarrow MI', p' \rightarrow p, q' \rightarrow q'\]

\[<CA', WA', CO', MI' | \{p' | S | q' \}>\]

5.3.5.12. substitution

\[<CA, WA, CO, MI | \{p | S | q \} >\]

\[<CA[t/z], WA[t/z], CO, MI | \{p[t/z] | S | q \} >\]

provided \(z\) does not appear free in \(S, CO, MI\) or \(q\).

5.3.5.13. auxiliary variables

Let \(AV\) be a set of variables such that \(x \in AV\) implies that \(x\) appears in \(S'\) only in assignments \(y:=t\), where \(y \in AV\). Then, if no variable in \(AV\) appears free in \(q\), \(CO\) or \(MI\) and \(S\) is obtained from \(S'\) by deleting all assignments to variables in
AV:
\[
\{p \mid S \mid q \} \quad \{p \mid S \mid q \}
\]

5.3.5.14. labels
\[
\{p \mid S \mid q \} \quad \{p \mid S \mid q \}
\]
where \( S \) is obtained from \( S' \) by deleting every label and any bracket.

5.4. An example proof
Consider the program \([M_1 \parallel \cdots \parallel M_n]\) where

\[
M_1:: a_1(\#) : [b := \neg b \; ; \text{while } \neg b \text{ do wait od} ; \text{if } x = 0 \text{ then } x := 1 \text{ else } x := 2 \; \text{fi}]
\]

\[
a_2(\#) : [b := \neg b ; \text{if } x = 1 \text{ then } x := 0 \text{ else skip fi}]
\]

\[
: [b := \text{true}; x := 1]
\]

\[
M_2:: : \text{[call } M_1.a_1(\#)]
\]

\[
M_3:: : \text{[call } M_1.a_2(\#)]
\]

The formula \(<\emptyset, \emptyset, \emptyset, \text{true} \mid \{\text{true}\}[M_1 \parallel \cdots \parallel M_n][x = 1]>\) is valid.
We sketch its proof.

Bracketed sections, auxiliary variables and labels are introduced, as far they are needed, as follows (\(h_{11}, h_{12}, p, h_2, h_3\) are auxiliary variables):

\[
M_1:: a_1(\#) : [l_0, h_{11} := h_{11} + 1 ; l_1, > ; b := \neg b ; \text{while } \neg b \text{ do wait.} l_3 \text{ od};
\]

\[
\text{if } x = 0 \text{ then } x := 1 \text{ else } x := 2 \; \text{fi}; l_4, < ; h_{11} := h_{11} + 1]
\]

\[
a_2(\#) : [l_5, \text{skip}; l_6, > ; b := \neg b ; \text{if } x = 1 \text{ then } x := 0 \text{ else skip fi}; l_7, < ; h_{12} := 0]
\]

\[
: [b := \text{true}; x := 1]
\]

\[
M_2:: : [l_8, < ; h_2 := 1 ; l_9, \text{call } M_1.a_1(\#); h_2 := 2 ; l_{10}, > ]
\]

\[
M_3:: : [l_{11}, < ; \text{skip}; l_{12}, \text{call } M_1.a_2(\#); h_3 := 0 ; l_{13}, > ]
\]

Define \(CA_i, WA_i, CO_i, MI_i\) \((i = 1, 2, 3)\) and GI as follows:

\[
CA_1 \equiv \{l_1 ; h_{11} = 1 \wedge ic_{11} = 1, l_6 ; h_{12} = 1\}
\]

\[
WA_1 \equiv \{l_3 ; MI_1 \wedge CA_1(l_1)\}
\]

\[
CO_1 \equiv \{l_2 ; CA_1(l_1), l_4 ; MI_1[h_{11} + 1 / h_{11}], l_7 ; MI_1[0 / h_{12}]\}
\]

\[
MI_1 \equiv x = (h_{11} + 2h_{12})/2 \wedge (h_{12} = 1 \rightarrow h_{11} \leq 1) \wedge (h_{11} = 0 \rightarrow (h_{12} = 0 \leftrightarrow b)) \wedge
\]

\[
(h_{11} = 1 \rightarrow (h_{12} = 0 \leftrightarrow b)) \wedge 0 \leq h_{12} \leq 1
\]

\[
CA_2 \equiv WA_2 \equiv \emptyset
\]

\[
CO_2 \equiv \{l_8 ; h_2 := 0 ; l_9 ; h_2 = 1\}
\]

\[
MI_2 \equiv h_2 = 1
\]

\[
CA_3 \equiv WA_3 \equiv \emptyset
\]
The specifications of the entry procedures have been fixed by these definitions. The specifications of the initial statements are:

\[
\begin{align*}
M_1 & : <C_{A_1}, W_{A_1}, C_{O_1}, M_{I_1} \mid h_{11} = 0 \land h_{12} = 1 \mid b := \text{true}; x := 1 \mid [M_I]>, \\
M_2 & : <C_{A_2}, W_{A_2}, C_{O_2}, M_{I_2} \mid h_2 = 0 \mid [\cdots] h_2 = 2 >, \\
M_3 & : <C_{A_3}, W_{A_3}, C_{O_3}, M_{I_3} \mid h_3 = 1 \mid [\cdots] h_3 = 0 >.
\end{align*}
\]

The proofs of these specifications are straightforward and are left to the reader. We show one interference freedom and one cooperation condition.

Interference freedom of \(W_{A_1}(l_3)\) w.r.t. the body of \(a_2\)

Let \(W_{A_1} \equiv \{l_6; W_{A_1}(l_3) \mid \text{ic} \geq 2 , l_3; W_{A_1}(l_3) \mid \text{ic} \leq 2 \}\) and \(C_{O_1} \equiv \{l_2; W_{A_1}(l_3)\}\)

To show \(<C_{A_1}, W_{A_1}, C_{O_1}, \text{true} \mid [M_{I_1}]; a_1(\#); [\cdots] [W_{A_1}(l_3)]>\).

We want to apply the procedure rule with the following premisses:

1. \(<\cdots \mid [M_{I_1} \land W_{A_1}(l_3)] \land \text{skip}[W_{A_1}(l_3)]>)>
2. \(<\cdots \mid [W_{A_1}(l_3)]; b := \neg b \mid \text{if } x = 0 \text{ then } x := 1 \text{ else skip } fi; [W_{A_1}(l_3) [0 / h_{12}])>)>
3. \(<\cdots \mid [W_{A_1}(l_3) [0 / h_{12}]) h_{12} := 0 [W_{A_1}(l_3)]>)>.

All three premisses are easily proved, and hence this interference condition holds.

The other specification, showing that \(W_{A_1}(l_3)\) is interference free with respect to the actions of \(a_1\) is trivialized by having \(W_{A_1}(l_3) \rightarrow \text{ic} \leq 1\) and \(C_{A_1}(l_1) \rightarrow \text{ic} \leq 1\), since the specification to be proved allows us to assume that \(\text{ic} \geq 2\).

So, we can use the module rule to derive

\(<C_{A_1}, \emptyset, C_{O_1}, M_{I_1} \mid h_{11} = 0 \land h_{12} = 1 \mid M_{I_1}[M_{I_1}]>)>.

The proofs of the module specifications of \(M_2\) and \(M_3\) are even more easy.

Cooperation between \(M_1\) and \(M_2\)

To show:

\(<\emptyset, \emptyset, \emptyset, \text{true} \mid [C_{O_2}(l_8) \land M_{I_1} \land G_I] h_2 := 1; \text{ic} := \text{ic} + 1; h_{12} := h_{12} + 1 \mid (C_{A_1}(l_1) \land G_I)>)>

and

\(<\emptyset, \emptyset, \emptyset, \text{true} \mid [C_{O_2}(l_9) \land C_{O_1}(l_4) \land G_I] h_{12} := h_{12} + 1; \text{ic} := \text{ic} - 1; h_2 := 2 \mid (C_{A_2}(l_{10}) \land G_I)>)>

These are trivial proofs, too. The only interesting thing is to observe that, since \(C_{O_2}(l_8) \rightarrow h_2 = 0\) and \(G_I \rightarrow h_{11} = h_2 \land (h_2 = 1 \rightarrow \text{ic} = 1)\), the assumption of \(C_{A_1}(l_1)\) about coexisting instances (\(\text{ic} = 1\)) is discharged.

This allows the program rule to be applied. Finally, applying the consequence, substitution, auxiliary variable and label rules (see section 5.3.5) gives
A nontrivial example proof concerning the Dutch national torus, has been given in [GdeRR82b]. The problem, “inspired by the Dutch National Flag”, originally appeared in Dijkstra’s [EWD608] in a version written in CSP.

A correctness proof of a distributed priority queue has been given in [GdeRR82a], concerning a program given in Brinch Hansen’s original CM article [BH78]. When trying to prove this program an error was discovered, which was subsequently mended.

6. CONCLUSIONS

This paper served two purposes. Firstly, we developed an axiomatization for a monitor-like language, Communicating Modules, based on the notions of cooperation ([AFdeR80]) and interference freedom ([OG76]). Cooperation is used to characterize the communication interaction between modules. Interference freedom characterizes the internal execution of each module. We interpret this as the non-deterministic interleaving of actions of a shared variable language. There is a two-fold generalization over ordinary shared variable languages here, since (1) in a module at any time the interleavable actions are taken from the then active procedure instantiations and (possibly) from the initial statement and (2) interleavable actions are not syntactically determined but are “bracketed” in time by the acquiring and relinquishing of the monitor-lock by a calling module. The variables of a module are shared between its initial statement and its procedure instantiations. It is this conceptual novelty that lead to a sound and relatively complete axiomatization ([G86]).

Secondly, we investigated how cooperation and interference freedom could be formulated in a less “algorithmic” way. The usual formulations rely on proof outlines rather than on the specifications to be shown valid. We showed that these notions do support syntax-directed proof systems. In this, we deliberately did not follow [A84] and [St86], who changed their notion of derivation in order to acquire syntax-directedness, but rather changed our notion of specification to one in which all assumptions on which validity of the specification depend are made explicit. This results in specifications in which given certain assumptions about how its environment interacts, a statement or module is specified to commit itself to certain behaviour. As such, the formalism conforms to the verification methodology as espoused by J. Mishra and M. Chandy ([MC81]) and as formalized in [ZdeRE85]. It was one of our specific aims to see how close one could come to such compositional proof systems while still using the notions of cooperation and interference freedom.

Because of this, we still have to alter the program text by introducing assignments to auxiliary variables and bracketed sections. Also, labels have to be used in order to associate assumptions and commitments to specific points in the program. These labels are not integrated with the assertion language. It is in fact possible to fully integrate them, as is shown in [G84] for a simpler language. Finally we mention that [HdeR86] traces, for CSP, the additional steps that have to be taken in order to obtain a truly compositional proof system.
REFERENCES


Compositional semantics for real-time distributed computing (Inf.&Control 1987)

Full abstraction of a real-time denotational semantics for an OCCAM-like language

A compositional proof theory for real-time distributed message passing

Questions to Robin Milner - A responder's commentary (IFIP86)

A timed failure semantics for communicating processes

Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)

Specifying passing systems requires extending temporal logic

On the existence of sound and complete axiomatizations of the monitor concept

Federatieve Databases

A formal approach distributed information systems

Delay-insensitive codes - An overview
Available Reports from the Theoretical Computing Science Group

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIR83.1 R. Koymans, J. Vytopil, W.P. de Roever</td>
<td>Real-Time Programming and Synchronous Message passing (2nd ACM PODC)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR84.1 R. Gerth, W.P. de Roever</td>
<td>A Proof System for Concurrent Ada Programs (SCP4)</td>
<td></td>
</tr>
<tr>
<td>TIR84.2 R. Gerth</td>
<td>Transition Logic - how to reason about temporal properties in a compositional way (16th ACM FOCS)</td>
<td></td>
</tr>
<tr>
<td>TIR85.1 W.P. de Roever</td>
<td>The Quest for Compositionality - a survey of assertion-based proof systems for concurrent programs, Part I: Concurrency based on shared variables (IFIP85)</td>
<td></td>
</tr>
<tr>
<td>TIR85.2 O. Grünberg, N. Francez, J. Makowsky, W.P. de Roever</td>
<td>A proof-rule for fair termination of guarded commands (Inf.&amp; Control 1986)</td>
<td></td>
</tr>
<tr>
<td>TIR85.3 F.A. Stomp, W.P. de Roever, R. Gerth</td>
<td>The $\mu$-calculus as an assertion language for fairness arguments (Inf.&amp; Control 1987)</td>
<td></td>
</tr>
<tr>
<td>TIR85.4 R. Koymans, W.P. de Roever</td>
<td>Examples of a Real-Time Temporal Logic Specification (LNCS207)</td>
<td></td>
</tr>
<tr>
<td>TIR86.1 R. Koymans</td>
<td>Specifying Message Passing and Real-Time Systems (extended abstract)</td>
<td>CSN86/01</td>
</tr>
<tr>
<td>TIR86.2 J. Hooman, W.P. de Roever</td>
<td>The Quest goes on: A Survey of Proof Systems for Partial Correctness of CSP (LNCS227)</td>
<td>EUT-Report 86-WSK-01</td>
</tr>
</tbody>
</table>
On Proving Communication Closedness of Distributed Layers (LNCS236)

Compositional Semantics for Real-Time Distributed Computing (Inf.&Control 1987)

Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language

A Compositional Proof Theory for Real-Time Distributed Message Passing

Questions to Robin Milner - A Responder's Commentary (IFIP86)

A Timed Failure Semantics for Communicating Processes

Proving Monitors Revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)

Specifying Message Passing Systems Requires Extending Temporal Logic

On the existence of sound and complete axiomatizations of the monitor concept