A PROOF SYSTEM FOR PARTIAL CORRECTNESS
OF DYNAMIC NETWORKS OF PROCESSES

Job Zwiers
Arie de Bruin
Willem Paul de Roever

RUU-CS-83-15
November 1983
A PROOF SYSTEM FOR PARTIAL CORRECTNESS
OF DYNAMIC NETWORKS OF PROCESSES

Job Zwiers
Arie de Bruin
Willem Paul de Roever

Technical Report RUU-CS-83-15
November 1983

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands
This paper will appear as extended abstract in:

A PROOF SYSTEM FOR PARTIAL CORRECTNESS
OF DYNAMIC NETWORKS OF PROCESSES

(Extended abstract)

Job Zwiers (*)
Arie de Bruin (**) Willem Paul de Roever (***)

(*) Department of Computer Science, University of Nijmegen,
Toernooiveld, 6525 ED Nijmegen, the Netherlands

(**) Faculty of Economics, Erasmus University,
P.O. Box 1738, 3000 DR Rotterdam, the Netherlands

(***) Department of Computer Science, University of Nijmegen,
and Department of Computer Science, University of Utrecht,
P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

Introduction

A dynamically changing network is a set of processes, executing in parallel and communicating via interconnecting channels, in which processes can expand into subnetworks. This expansion is recursive in the sense that the so formed subnetwork can contain new copies of the expanding process. After all component processes of a subnetwork have terminated, the expanded process contracts (shrinks) again and continues its execution. We define a simple language, called dynamic CSP, which can describe such networks. We introduce a formal proof system for the partial correctness of dynamic CSP programs. The proof system is built upon a new type of correctness formulae, inspired by Misra and Chandy [MC], which allow for modular specifications c.q. proof of properties of a network. In the full paper, the proof system is shown to be sound with respect to a denotational semantics for the language.

Acknowledgements

We are indebted to many people for their helpful comments. We would like to thank P. van Emde Boas, J.W. de Bakker and especially R. Gerth for clarifying discussions.

---------

The major portion of the research reported in this paper was carried out at the University of Utrecht as part of the requirements for obtaining a "doctoraal examen" by the first author.
1. The language

First we give the contextfree syntax of dynamic CSP.

In this syntax, x and u stand for program variables, D and E for channel variables, P for procedure names, s for expressions and b for boolean expressions.

**Statements:**

\[
S ::= x := s \mid \text{skip} \mid b \mid D ? x \mid D ! s \mid S_1 ; S_2 \mid [S_1 \sqcap S_2] \mid \text{cobegin } N \text{ coend } \mid P(E_{in-1}, \ldots, E_{in-k}; E_{out-1}, \ldots, E_{out-1}; u_1, \ldots, u_m)
\]

**Networks:**

\[
N ::= S_1 || S_2
\]

Procedure declarations:

\[
T ::= P(D_{in-1}, \ldots, D_{in-k}; D_{out-1}, \ldots, D_{out-1}; x_1, \ldots, x_m) \text{ begin } S \text{ end}
\]

Programs:

\[
R ::= T_1, \ldots, T_n : \text{ begin } S \text{ end}
\]

The intuitive meaning of \(x := s\), \text{skip} and \(S_1 ; S_2\) should be clear. \([S_1 \sqcap S_2]\) stands for nondeterministic choice between \(S_1\) and \(S_2\). Boolean expressions are incorporated as statements. They function as "guards": whenever \(b\) evaluates to "true", the guard can be passed, i.e. it is equivalent to "\text{skip}" in this case. When \(b\) evaluates to "false" the guard cannot be passed and the computation is aborted. Because we are only interested in partial correctness ("...if a program reaches \((a)\) certain point\((s)\) then..."), a more familiar construct as \(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}\) can be expressed in our language as \(b; S_1 \sqcap \neg b; S_2\). We will freely use such "derived" constructs in our examples. A network \(S_1 || S_2\) calls for concurrent execution of \(S_1\) and \(S_2\). In such a network, \(S_1\) and \(S_2\) are not allowed to have "shared" program variables. The two component processes of a network can communicate with each other (only) along named, directed channels. Communication along a channel, say \("D\) occurs when an output command "\(D! s\)" is executed by one of the component processes simultaneously, i.e. synchronized, with an input command "\(D ? x\)" of the other one. The value of \(s\) is then assigned to the program variable \(x\) and both processes continue their execution. In dynamic CSP, channels always connect exactly two processes. So a process cannot both read from and write to one and the same channel, nor are two different processes allowed to both read from or both write to some common channel. A channel from which some process reads or to which it writes is called an external input or output channel of that process, respectively. When two processes are bound together into a network, their common channels, along which they communicate, are said to be internal channels of that network. The concepts of internal and external channels are important to the modularity of our proof system. When dealing with "nested" networks, i.e. networks as \(S_1 || S_2\) in which \(S_1\) and \(S_2\) are themselves (or contain) subnetworks, it is possible that some subnetwork has an internal channel with the same name as some channel of the main network. This is even unavoidable when the subnetwork and the main network belong to different incarnations of the same procedure body in case of a recursive procedure call. Such channel name "clashes" are resolved by introducing a kind of block structure with the \text{cobegin - coend} construct, which "hides" internal channels, i.e. no internal channel of \(S_1 || S_2\) is visible anymore in the process \text{cobegin } S_1 || S_2 \text{ coend}.\]
So in \emph{S} = \texttt{cobegin} \texttt{S}_1 \parallel \texttt{cobegin} \texttt{S}_2 \parallel \texttt{S}_3 \texttt{coend coend}

with \emph{S}_1 = \texttt{D?x}, \emph{S}_2 = \texttt{D!0}, \emph{S}_3 = \texttt{D?y},

the \emph{S}_2 process communicates with \emph{S}_3 along the \emph{D} channel internal to \emph{S}_2∥\emph{S}_3, and not with \emph{S}_1. The \emph{D} channel of \emph{S}_1 is an external input channel of \emph{S}. We note that when no "clashes" arise between the external channel names of processes \emph{S}_1, \emph{S}_2 and \emph{S}_3, then the semantic operator for parallel composition is associative for the network consisting of \emph{S}_1, \emph{S}_2 and \emph{S}_3 executing concurrently, so we can write

\texttt{cobegin} \texttt{S}_1 \parallel \texttt{S}_2 \parallel \texttt{S}_3 \texttt{coend}

without (semantic) ambiguity. In agreement with the modular character of dynamic CSP, we have for recursive procedures a scope concept different from that of Algol like languages. All variables used in some procedure body (which is bracketed by \texttt{begin - end}) are assumed to be \texttt{local} variables, i.e. there are no references to any kind of "global" variables possible. (Correspondingly, there is no explicit variable declaration mechanism needed in dynamic CSP.) The parameter list of a procedure consists of input channels, followed by output channels, followed by value/result-variable parameters. To simplify matters technically, we impose the restriction that all names in a (formal or actual) parameter list be distinct. This avoids any kind of "aliasing" that could introduce unwanted sharing of program- or channel variables by two processes. This section is concluded by an example of an algorithm known as a "priority queue".

\begin{verbatim}
Q(in;out;)
begin
  shrink := false;
  while \texttt{true} \shrink
    do
      [ in?val + \texttt{cobegin}
        P(in,int1;out,int2;val)
        \parallel
        Q(int2;int1;)
      \texttt{coend}
      \texttt{out}("**") + shrink := true
    od
end,

P(lin,rin;out,rout;ownval)
begin
  shrink := false;
  while \texttt{true} \shrink
    do
      [ lin?newval + largest := max(ownval,newval);
        ownval := min(ownval,newval);
        rout!largest
        \texttt{endif}
        lout!ownval + rin?ownval;
        shrink := (ownval="*")
    od
end : 
begin
  \texttt{cobegin} Q(D;E;) \parallel \texttt{Userproc}(E;D; \texttt{coend})
end
\end{verbatim}
The queue can hold an arbitrary number of values: it expands or shrinks into as many processes as are needed. It can be sent a value along an input channel "in", or requested for a value along an output channel "out". In the latter case it sends the least value currently in the queue, if any.

fig.1 Expansion, corresponding to the \texttt{cobegin - coend} part of a "Q" process.

The "queue" starts as a single Q process. When sent a value, this Q process expands into a P process, which holds the value in its variable "ownvalue", and a new copy of a Q process. (see fig.1) When this expanded process is sent another value, this value is received by the internal P process, which compares it with its "ownvalue". It then keeps the smallest of the two, sending the larger one to its right neighbour which is, in this case, the internal Q process which will expand itself, etc. However, when a P process is asked for a value it sends its "ownvalue" which it is currently holding, and then tries to get a value from its neighbour. When this neighbour is itself a P process it acts analogously, but if it is a Q process it sends a "**" value to P and then terminates. The P process, upon receiving this "**", will also terminate, so the internal network consisting of these two processes terminates. The Q process which "envelopes" this terminated network can then receive a value and expand again, or it can be asked for a value, whereafter it sends a "**" and terminates itself too, etc....
fig.2 A queue holding the values 3, 7 and 9 is sent a value 5. This new value is inserted, causing the queue to expand. Before this process of insertion has come to an end, the queue is already asked for its least value. This causes a deletion of the "front" value (3); this deletion then "ripples" towards the end of the queue, after which the queue shrinks.
2. The proof method

By now, a number of proof systems has been designed for "static" networks consisting of a fixed number of concurrently executing processes. (e.g.:[AFR], [SD], [MC], [L]) With the proof systems of [AFR] a specification of a collection concurrently executing processes is derived by means of a "cooperation test" from proof outlines for the component processes. This derived specification is not of the same form as the specifications of the components! Because of this, we cannot repeat the procedure to derive some specification of a still larger network in which the collection processes mentioned above can be handled as just one component. Obviously, such a proof system cannot be used for dynamically evolving networks. Another type of proof system, for example that of [SD], does not distinguish between the form of the specifications of networks on the one hand and of their components on the other. The correctness formulae of the system of [SD] are expressed in terms of the notions "trace" and "state". A state is a function describing the current contents of the program variables of some process. A trace is a sequence of records of communication, indicating which values were communicated along which channel (or with which process in the case of CSP), and also showing the chronological order of these communications. The system of [SD] uses Hoare style correctness formulae and proof rules. A general disadvantage of the Hoare method is, that a Hoare style assertion itself is not suited to express properties about a process which is not intended to terminate. The only way seems to be to resort to proof outlines, i.e. program texts with assertions attached to intermediate control points. However, for a modular set up of a proof system, complete proof outlines are hardly satisfactory as program specifications. For a specification of a, possibly nonterminating, process, which will execute concurrently with, and communicate with some "unknown" collection of other processes, it seems plausible to include some kind of invariant to describe the interface of the process with its environment. This invariant must be expressed solely in terms of the (traces of) externally visible channels of the process, and not in terms of the internal state or internal channels. This leads to the following type of correctness formulae:

\[ I : \{p\} S \{q\} \]

where \( p \) is a precondition on the initial state and trace, and \( q \) is a postcondition on the final state and trace of \( S \). \( I \) is an trace invariant for computations of \( S \). The informal meaning of such a formula is:

If \( p \) and \( I \) hold initially (that is, \( p \) holds for the initial state and trace, \( I \) holds for the initial trace), then:
1. \( I \) is preserved during the computation of \( S \), that is, \( I \) still holds after any communication of \( S \).
2. If \( S \) terminates, the postcondition \( q \) will hold for the final state and trace of \( S \).

With this type of formulae, a proof rule for parallel composition can be envisaged as:

\[ I_1 : \{p_1\} S_1 \{q_1\} , I_2 : \{p_2\} S_2 \{q_2\} \]

\[ I_1 \& I_2 : \{p_1 \& p_2\} S_1 || S_2 \{q_1 \& q_2\} \]
The conclusion of this rule is called an *internal* specification of the network \( S_1 || S_2 \), because it will mention channels internal (as well as external) to \( S_1 || S_2 \). To turn an internal specification into an external one, we introduce the following abstraction rule:

\[
I' : \{ p' \} S_1 \ || \ S_2 \{ q' \} , \ I \Rightarrow I^
\]

\[
(p \land I \land \pi_{intchan} = A) \Rightarrow (p' \land I') , \ q' \Rightarrow q
\]

\[
I : \{ p \} \text{cobegin} S_1 \ || \ S_2 \text{coend} \{ q \}
\]

provided that the free channel names, used in \( I, p \) and \( q \) are included in the external channels of \( S_1 || S_2 \).

\( \pi_{intchan} \) stands for the projection of the trace of \( S_1 || S_2 \) onto the internal channels of \( S_1 || S_2 \), i.e. the subsequence of the trace formed by deleting all records of communication with a channel name not internal to \( S_1 || S_2 \).

\( A \) stands for the "empty trace".

The merit of the above rule for parallel composition is the simplicity of its form. Moreover, for merely "pipelined" processes, its usage in proofs is also straightforward as the following example shows.

Take the following processes:

\[
S_1 = \text{do IN} \times x \; \text{if} \; x > 0 \; \text{then} \; \text{D}!x \; \text{else} \; \text{D}!-x \; \text{fi} \; \text{od}
\]

\[
S_2 = \text{do D} ? y \; \text{OUT}! (\text{enter}(\sqrt{y})) \; \text{od}
\]

\[
S = \text{cobegin} S_1 \ || \ S_2 \text{coend}
\]

Denoting by:

- channel names like "D" the corresponding projection of the trace onto that channel.
- \( D[i] \): the \( i \)-th element of (projection) \( D \)
- \( |D| \): the length of (projection) \( D \)
- \( \text{trace}_1 \subseteq \text{trace}_2 \): that \( \text{trace}_1 \) is some initial prefix of \( \text{trace}_2 \)
- \( f(\text{trace}) \) for some function \( f \): the trace formed by applying \( f \) componentwise

then, we can write down the following specification for \( S \):

\[
\text{OUT} = \text{enter}(\sqrt{\text{abs}(\text{IN}))} : \{ \text{IN} = \text{OUT} = A \} \; S \{ \text{true} \}
\]

To prove this from specifications for \( S_1 \) and \( S_2 \), we can choose these last ones as:

\[
D = \text{abs}(\text{IN}) : \{ \text{IN} = \text{OUT} = A \} \; S_1 \{ \text{true} \}
\]

and

\[
\text{nonneg}(D) \Rightarrow \text{OUT} = \text{enter}(\sqrt{D}) : \{ D = \text{OUT} = A \} \; S_2 \{ \text{true} \}
\]

(where \( \text{nonneg}(D) \) is some predicate expressing that all values occurring in the trace of \( D \) are nonnegative ones.)

From these two, the desired specification of \( S \) follows easily with the above two rules.

*Heuristically*, we can regard the invariants of \( S \) and of \( S_1 \) as *commitments* about the behaviour of \( S \) and \( S_1 \), respectively. The invariant of \( S_2 \) is split into a "commitment" \( \text{OUT} = \text{enter}(\sqrt{D}) \) which is preserved if and only if the *assumption* \( \text{nonneg}(D) \) about the environment is not violated. In the network \( S \), the commitment of \( S_1 \) implies the assumption of \( S_2 \). This distinction between assumption and commitment will be formalized after our next example. This example shows that the usage of
the above rule for parallel composition is rather cumbersome for non-"pipelined"
networks, in which information flows back and forth between the component processes
rather than in one direction.

We take:

\[ S_1 = x := 0 ; \text{do } D! (x+1) ; \text{E?} x \text{ od} \]
\[ S_2 = \text{do } D? y ; \text{E!(y+1)} ; \text{OUT!} y \text{ od} \]
\[ S^T = \text{opbegin } S_1 \parallel S_2 \text{ coend} \]

and try to prove:

\[ \text{OUT} \leq (1,3,5,...) ; [\text{OUT} = A] S \{\text{true}\} \]

from specifications for \( S_1 \) and \( S_2 \).

\( \{\text{OUT} \leq (1,3,5,...) \} \) can be written more formally as:

\[ \forall j \left[ \text{OUT}^j \Rightarrow \text{OUT}^j = 2j-1 \right] \]

Again, heuristically, \( S_1 \) has a commitment \( C_1 \equiv D \leq (1,3,5...) \)
which depends on the assumption \( A_1 \equiv E \leq (2,4,6,...) \).
For \( S_2 \), we have a commitment \( C_2 \equiv E \leq (2,4,6,...) \) & \( \text{OUT} \leq (1,3,5...) \)
which depends on the assumption \( A_2 \equiv D \leq (1,3,5...) \)

Unfortunately, the specifications:

\[ A_1 \Rightarrow C_1 : \{D = E = A\} S_1 \{\text{true}\} \]
\[ A_2 \Rightarrow C_2 : \{D = E = \text{OUT} = A\} S_2 \{\text{true}\} \]

will not suffice to deduce the specification for \( S \), as we are stuck on "circular"
reasoning when we apply the parallel composition rule:

\[ C_1 \Rightarrow A_2 \Rightarrow C_2 \Rightarrow A_1 \Rightarrow C_1 \Rightarrow .... \]

Some adequate specifications, which are however somewhat difficult to appreciate, are:

\[ D[1] = 0 \& \forall i \left[ (1 \leq i \leq D) \& E[i-1] = 2i-2 \Rightarrow D[i] = 2i-1 \right] ; \{D=E=A\} S_1 \{\text{true}\} \]
\[ \forall i \left[ (1 \leq i \leq \text{OUT}) \& D[i] = 2i-1 \Rightarrow (E[i] = 2i \& \text{OUT}[i] = 2i-1) ; \{D=E=\text{OUT}=A\} S_2 \{\text{true}\} \right] \]
\[ \{D = E = \text{OUT} = A\} S_2 \{\text{true}\} \]

From the conjunction of the two invariants we can, by inductive reasoning, derive an
invariant of \( S_1 \parallel S_2 \) from which the desired invariant of \( S \) can be deduced with the
abstraction rule; the "circular" reasoning is turned into a "spiral". We can para-
phrase the specification for, say \( S_1 \), as follows:

Let precondition \( p \equiv [D = E = A] \) and \( C_1 \) (as above) hold initially, then if \( A_1 \) holds
for a prefix of the trace of a computation of \( S_1 \), the commitment \( C_1 \) holds for that
prefix extended with the next communication.

Generalizing this pattern, we now introduce a new type of correctness formulae, in-
spired by a similar type of formulae used by Misra and Chandy [MC].

Consider the formula:

\[ (A,C) : \{p\} S \{q\} \]

Informally, this formula means the following:
Assume p & C holds for the initial state and trace, then the following two conditions hold:

1. If A holds after each communication of S up to a certain point, then C holds after each communication of S up to that point and also after the next communication of S, if present.
2. If S terminates, then if A holds after each communication of S, then the postcondition q holds for the final state and trace.

The pair (A,C) is called the assumption-commitment pair. The syntactic restrictions on assumptions and commitments are that they only refer to the trace of computations of S, and not to the states S passes through.

Also, the only way to refer to traces in assumptions, commitments, pre- or postconditions is by means of projections Π_cset, with cset a set of channel names. Such a projection stands for the subsequence of the trace, formed by deleting all records of communication which do not represent a communication along one of the channels in cset. When cset contains just one channel name, we use this name as an abbreviation for the projection Π_{cset}.

Due to our new correctness formulae, we can formulate a new proof rule which makes the explicit inductive reasoning in the example above unnecessary, as this induction becomes implicit in its soundness. This rule, called the network rule, is a slight modification of a rule proposed by Misra and Chandy [MC], and replaces the rule for parallel composition introduced above.

The network rule is:

\[(A_1, C_1) : \{p_1\} S_1 [q_1], (A_2, C_2) : \{p_2\} S_2 [q_2] \quad (1)\]

\[A_{\text{net}} & C_1 \Rightarrow A_2, \quad A_{\text{net}} & C_2 \Rightarrow A_1 \quad (2)\]

\[A_{\text{net}}, C_1 & C_2) : \{p_1 & p_2\} S_1 || S_2 [q_1 & q_2]\]

with the restrictions:

\[\text{free}(A_1, C_1, p_1, q_1) \subseteq \text{free}(S_1)\]

\[\text{free}(A_2, C_2, p_2, q_2) \subseteq \text{free}(S_2)\]

(Where free(X) denotes the free program- and channel variables of X. The free channels of a process are its external channels.)

Example: in the new formalism, we can express specifications for the processes in the example above as:

\[(A_1, C_1) : [D = E = A] S_1 \{\text{true}\}\]
\[(A_2, C_2) : [D = E = \text{OUT} = A] S_2 \{\text{true}\}\]

\[[\text{true}, C_1 & C_2) : [D = E = \text{OUT} = A] S_1 || S_2 \{\text{true}\} \quad (*)\]

\[[\text{true}, C) : [\text{OUT} = A] \begin{align*}
\text{co}\begin{align*}
\text{begin} & S_1 || S_2 \text{co}\text{end} \{\text{true}\} \quad (**)
\end{align*}
\end{align*}\]

with:

\[A_1 = E \leq (2, 4, 6, ...)\]
\[A_2 = C_1 = D \leq (1, 3, 5, ...)\]
\[C_2 = A_1 \& \text{OUT} \leq (1, 3, 5, ...)\]
\[C = \text{OUT} \leq (1, 3, 5, ...)\]

Clearly, (*) can be derived from the specifications for S_1 and S_2 by means of the
network rule. Finally, an application of the abstraction rule yields (**).

A formal soundness proof of the network rule can be found in the full paper. A sketch of this proof is as follows:

Assume that $C_1$ & $C_2$ & $p_1$ & $p_2$ holds initially.

Then we have to prove:

1. If $A_{\text{net}}$ holds after communications up to a certain point, then $C_1$ & $C_2$ holds up to that point and also after the next communication.

2. If $A_{\text{net}}$ holds after all communications up to termination, then $q_1$ & $q_2$ holds for the final state and trace.

Here, we only show (1). So assume also that $A_{\text{net}}$ holds after communications up to (and including) a certain point, say $X$, in the execution. We prove, by induction, that for all points $Y$, preceding and including $X$, the following holds:

(*) $A_1, A_2, C_1$ and $C_2$ hold after communications up to and including $Y$.

(**) $C_1$ and $C_2$ hold also after the first communication after $Y$, say at $Y'$.

(a) Initially, (*) is trivially fulfilled, since no communication took place. (And we only require something to hold after communication). Now for (**), we know by assumption that $p_1$ & $C_1$ and $p_2$ & $C_2$ hold initially.

Now there are three cases:

- $S_1$ communicated along an external channel of the network.
- $S_2$ communicated along an external channel of the network.
- Both $S_1$ and $S_2$ communicated, along an internal channel of the network.

Clause (1) of the network rule guarantees that the commitments of the processes which actually communicated hold after this communication. The fact that we allow references to traces in correctness formulae only by means of projections onto sets of channels can be used to show that the commitment of a process is preserved also when the process did not participate in the communication. So in all cases, $C_1$ and $C_2$ hold after the first communication.

(b) Now fix some arbitrary point $Y$ after the first communication. By assumption, $A_{\text{net}}$ holds for $Y$. Also, by induction we may assume that $C_1$ and $C_2$ hold for $Y$. But then, with clause (2) of the network rule, $A_1$ and $A_2$ hold for $Y$, which establishes (*). Finally, from (*), the induction hypothesis and the assumption that $p_1$ & $C_1$, and $p_2$ & $C_2$ hold initially, it follows with clause (1) again that $C_1$ and $C_2$ hold after the first communication, for $Y'$.

3. The proof system

There are several points which deserve attention. Firstly, since recursive procedures are included in our language, statements and correspondingly correctness formulae, have a meaning only in the context of some environment consisting of procedure declarations.

So formulae in our system are of the form:

\[
\langle \text{Decl} \mid (A,C) : \{p\} S \{q\} \rangle
\]
(Where Decl is a set of procedure declarations)

However, apart from the recursion rule, we have that for each (axiom or) rule the
environments mentioned in premisses and conclusion of the rule are all the same.
Therefore, in almost all cases, we do not write down environments explicitly in the
formulae, assuming that the environment is clear from context.

Secondly, although in this abstract we have not defined explicitly the assertion
languages to be used in formulae for assumptions, commitments, pre- and postcondi-
tions, we remark here that these languages, besides program- and channel variables,
also include "logical" or "freeze" variables as they are called in the literature.
Such freeze variables cannot occur in the program text, so the value they denote is
not affected by the computations of the program. Correctness formulae containing
free freeze variables are implicitly assumed to be universally quantified for these
variables, so:

\[(A,C) : [p] S [q]\]

with free freeze variable \(f\), means the same as:

\[\forall f [ (A,C) : [p] S [q] ]\]

We now discuss in what sense "classical" Hoare style formulae like the normal as-
signedment axiom are incorporated in our system. Notice that if "S" is some noncom-
municating statement, then validity of

\[(A,C) : [p] S [q]\]

boils down to: if \(p \land C\) holds for the initial state and trace, then \(q\) holds for the
final state and trace. So whenever the Hoare formula \([p] S [q]\) is valid (meaning
that, if \(p\) holds for the initial state and trace, then \(q\) holds for the final state and trace), then \((A,C) : [p] S [q]\) is valid in our system for every assumption-
commitment pair \((A,C)\). We therefore introduce axiom schemes \([p] S [q]\) in our sys-
tem, from which normal axioms (schemes) can be obtained by adding some arbitrary
\((A,C)\) pair in front of it. It will now be clear that we can incorporate all normal
Hoare axioms provided we regard them as schemes in the above sense. For our partic-
ular language, we have the following axiom schemes:

\[
[p[s/x]] \ x:=s \ [p] \quad \text{(assign)}
\]

\[
(p[s/x]) \text{ denotes the assertion } p \text{ in which } s \text{ is subtituted for free occurrences}
\text{ of } x
\]

\[
[p] \ skip \ [p] \quad \text{(skip)}
\]

\[
[p] \ b \ [p \land b] \quad \text{(test)}
\]

However, these axioms are not sufficient if we want a complete proof system! For
instance, look at: \((true,false) : [true] \ skip \ [false]\).

This formula is valid, but it cannot be derived from our axioms, since:
\((true) \ skip \ [false]\) is not valid if regarded as a classical Hoare formula. To
correct this we introduce the following rule:
Now, what about "classical" Hoare rules like the sequential composition rule? Here we introduce proof rule schemes like for example:

\[(p_1) S_1 \{q_1\}, (p_2) S_2 \{q_2\}\]

\[p_3 S_3 \{q_3\}\]

which is valid if and only if

\[(A,C) : (p_1) S_1 \{q_1\}, (A,C) : (p_2) S_2 \{q_2\}\]

\[(A,C) : (p_3) S_3 \{q_3\}\]

is valid for every feasible (A,C) pair.

(notice that the same (A,C) pair is used in all formulae of the premises as well as of the conclusion of the rule!)

We now claim that the following scheme is sound in our system:

\[(p) S_1 \{r\}, (r) S_2 \{q\}\]

\[(p) S_1 ; S_2 \{q\}\]

(sequential composition)

( So the normal Hoare style sequential composition rule can be regarded as a scheme in our system.)

We give a sketch of the soundness proof of this scheme: Take some arbitrary pair (A,C), and assume that p & C holds initially. We prove that if \(A\) holds after communications up to some point in the computation of \(S_1 ; S_2\), then C holds after communications up to this point and also after the next communication, if present. (The proof that q holds upon termination if A holds after all communications of \(S_1 ; S_2\) follows similar lines.) So assume also that A holds up to some point, say X. The only interesting case is when X lies in the midst of the execution of \(S_2\). In this case, A holds after all communications of \(S_1\), so with the premiss for \(S_1\), we know that C holds after communications of \(S_1\), and that r and C hold when \(S_1\) terminates. But then the premiss for \(S_2\) guarantees that C also holds after communications of \(S_2\) up to X and after the next communication of \(S_2\).

It turns out that many "classical" Hoare rules remain sound when regarded as proof rule schemes in our system. In particular, for dynamic CSP we have besides the sequential composition rule the following rule for the nondeterministic choice construct:

\[(p) S_1 \{q\}, (p) S_2 \{q\}\]

\[(p) [ S_1 \cup S_2 ] \{q\}\]

(choice)

We continue with rules for input and output commands. In these rules, we use substitutions in assertions like: A[D<ν>/D] denoting the assertion formed from A by replacing terms \(\Pi_{cset}\), with D ∈ cset, by \(\Pi_{cset}<D,ν>\). As usual, care must be taken that the variable ν not becomes bound by some quantifier in A. (In the assertion
language, two consecutive trace expressions denote the concatenation of the two corresponding traces. A term \( \langle D, v \rangle \) denotes the one element trace, indicating the communication of a value \( v \) along the channel \( D \); when the channel is clear from context, the abbreviation \( \langle v \rangle \) is also used for such a term.

\[
C \; p \Rightarrow \forall v : ( C[D<v>/D] ) \quad \text{(input)}
\]

\[
C \; p \Rightarrow \forall v : ( A[D<v>/D] \Rightarrow q[D<v>/D,v/x] )
\]

\[
(A,C) : \{ p \} \; D?x \; \{ q \}
\]

\[
C \; p \Rightarrow C[D<s>/D] \quad \text{(output)}
\]

\[
C \; p \Rightarrow ( A[D<s>/D] \Rightarrow q[D<s>/D] )
\]

\[
(A,C) : \{ p \} \; D!s \; \{ q \}
\]

The first premiss of both rules is clear: we must show that \( C \) is preserved for this communication. Since we do not know which value we are going to receive, we must show this for every possible value for the case of an input command. The second premiss for the input rule is also clear: the assumption \( A \) can be used to derive some property of the value actually received. This property can be expressed in the postcondition of the command.

The role of the \( A \)-term in the second premiss of the output rule is less clear, since the communicated value is not unknown. However, in the next example it is used to fix the channel along which communication occurs:

\[
S_1 = [ D!0 \; ; \; E!0 \; ; \; x:=0 \; \diamond \; E!0 \; ; \; D!0 \; ; \; x:=1 ]
\]

\[
S_2 = E?q ; \; D?q
\]

\[
S = \text{cobegin} \; S_1 \; || \; S_2 \; \text{coend}
\]

We can prove: \( (true,true) : \{ true \} \; S \; \{ x=1 \} \)

from:

\[
(A_1,\text{true}) : \{ \Pi_{D,E} = A \} \; S_1 \; \{ x=1 \}
\]

and:

\[
(\text{true},C_1) : \{ \Pi_{D,E} = A \} \; S_2 \; \{ \text{true} \}
\]

with:

\[
A_1 = C_2 = \exists v,w : ( \Pi_{D,E} \leq (\langle E,v,D,w \rangle) )
\]

To prove the formula for \( S_1 \) we must show among others that:

\[
(A_1,\text{true}) : \{ \Pi_{D,E} = A \} \; D!0 \; ; \; E!0 \; ; \; x:=0 \; \{ x=1 \}
\]

Here we can use the assumption \( A_1 \) in the output rule to obtain:

\[
(A_1,\text{true}) : \{ \Pi_{D,E} = A \} \; D!0 \; \{ \text{false} \}
\]

The rest of the proof now follows easily.

In the next two rules, for recursive procedure calls, we use the following notation:

\( \overline{D} \) and \( \overline{X} \) are abbreviations for lists of channel and program variables.
\( E/D \) denotes the substitution of the \( E \)-variables for corresponding \( D \)-variables.

\( \omega \) denotes some special value, used to initialize all local variables of a procedure body.

\( \text{var}(X) \) denotes the free program variables of \( X \).

\( \text{chan}(X) \) denotes the free channel variables of \( X \).

The rules are:

\[
\begin{align*}
\text{Decl} & \mid [p] \quad P(\overline{\text{in}}; \overline{\text{out}}; \overline{X}) \quad \{q\} & \quad \text{(recursion)} \\
\text{Decl} & \mid \{p & \quad \overline{Y} \Rightarrow \overline{W}\} \quad S_0 \quad \{q\} \\
\text{Decl} & \mid \{P(\overline{\text{in}}; \overline{\text{out}}; \overline{X}) \} \quad \text{begin} \quad S_0 \quad \text{end} \quad \| \quad \{p\} \quad P(\overline{\text{in}}; \overline{\text{out}}; \overline{X}) \quad \{q\} 
\end{align*}
\]

Provided that \( \text{var}(p,q) \subseteq \{\overline{X}\} \), and where \( \overline{Y} \) denotes the list of local variables of \( S_0 \) excluding the parameters \( \overline{X} \). (Notice that we used a proof rule scheme here)

The essence of the soundness proof for this rule is an induction on the recursion depth of a call of some procedure. A problem is that this recursion depth is not necessarily bounded for a given call. For formulae in which such nonterminating calls occur, the requirement that some postcondition holds upon termination is of course trivially fulfilled, but the requirement for the \((A,C)\) pair is not! The solution to this problem is to remember that the validity of correctness formulae is formulated in terms of (finite) prefixes of the trace of some process, and that each of these prefixes is produced already after the computation has reached some finite recursion depth. See the full paper for a real soundness proof.

\[
(A,C) \quad : \quad \{p\} \quad P(\overline{\text{in}}; \overline{\text{out}}; \overline{X}) \quad \{q\} \quad \text{(parameter substitution)}
\]

\[
(A[\bullet], C[\bullet]) \quad : \quad \{p[\bullet]\} \quad P(\overline{E}_{\text{in}}; \overline{E}_{\text{out}}; \overline{U}; \overline{X}) \quad \{q[\bullet]\}
\]

where \( [\bullet] = [\overline{E}_{\text{in}}; \overline{E}_{\text{out}}; \overline{U}; \overline{X}] \)

And provided that:

\[
\begin{align*}
\overline{E}_{\text{in}} \cup \overline{E}_{\text{out}} \cap \text{chan}(A,C,p,q) \subseteq (\overline{D}_{\text{in}}; \overline{D}_{\text{out}}) \\
\overline{U} \cap \text{var}(p,q) \subseteq \overline{X}
\end{align*}
\]

Due to our restriction that all names in a (formal or actual) parameter list must be distinct, and also to the absence of "global" variables, we could keep this rule quite simple.

The rules above can be used to derive properties of changes made to the actual parameters of some procedure call, but they are not sufficient to prove that other variables, not used as actual parameter, are left unchanged by this call. Similar problems arise when we try to prove that the trace of some channel \( D \) is left invariant by the execution of some network of which \( D \) is not an external channel. To be able to prove these invariance properties, we introduce an axiom and some rules.

\[
(A,C) \quad : \quad \{p\} \quad S \quad \{p\} \quad \text{(invariance)}
\]

Provided that \( \text{free}(A,C,p) \cap \text{free}(S) = \emptyset \)

\( \text{free}(X) \) denotes the free program- and channel variables of \( X \)

Soundness of this rule depends heavily on the fact that assertions can refer to traces only by means of projections on channels.
\[ (A_1, C_1) : \{p_1\} S \{q_1\}, (A_2, C_2) : \{p_2\} S \{q_2\} \quad \text{(conjunction)} \]
\[ (A_1 \& A_2, C_1 \& C_2) : \{p_1 \& p_2\} S \{q_1 \& q_2\} \]
\[ \{p\} S \{q\} \quad \text{(freeze variable substitution I )} \]
\[ \{p[e/f]\} S \{q[e/f]\} \]

Where \( f \) is some freeze variable and \( e \) is some expression, not containing program- or channel variables.

\[ \{p\} S \{q\} \quad \text{(freeze variable substitution II )} \]
\[ \{p[s/f]\} S \{q\} \]

Where \( f \) is some freeze variable and \( s \) is some expression, and provided that \( f \) does not occur free in \( q \).

We close this section with a restatement of the network rule, a reformulation of the abstraction rule, and the introduction of a consequence rule, which is used in connection with the other two.

\[ (A_1, C_1) : \{p_1\} S_1 \{q_1\}, (A_2, C_2) : \{p_2\} S_2 \{q_2\} \quad \text{(network)} \]
\[ A_{\text{net}} \& C_1 \Rightarrow A_2, A_{\text{net}} \& C_2 \Rightarrow A_1 \]
\[ (A_{\text{net}} \& C_1 \& C_2) : \{p_1 \& p_2\} S_1 \parallel S_2 \{q_1 \& q_2\} \]

With the restrictions:

\[ \text{free}(A_1, C_1, p_1, q_1) \subseteq \text{free}(S_1) \]
\[ \text{free}(A_2, C_2, p_2, q_2) \subseteq \text{free}(S_2) \]

\[ (A, C) : \{p \& \Pi_{\text{intchan}}^=A\} S_1 \parallel S_2 \{q\} \quad \text{(abstraction)} \]
\[ (A, C) : \{p\} \begin{array}{c} \text{cobegin} \\ \text{S_1} \\ \text{coend} \end{array} \{q\} \]

Where \text{intchan} denotes the internal channels of \( S_1 \parallel S_2 \), and provided that \( \text{chan}(A, C, p, q) \cap \text{intchan} = \emptyset \).

\[ (A', C') : \{p'\} X \{q'\} \quad \text{(With } X = S \text{ or } X = N) \]
\[ A \Rightarrow A', C \Rightarrow C, p \Rightarrow p', q \Rightarrow q, p \& C \Rightarrow C' \]
\[ (A, C) : \{p\} X \{q\} \]

The intended usage of these rules is as follows:

1. Use the network rule to derive a formula for \( S_1 \parallel S_2 \).
2. Use the consequence rule to remove all information about the internal functioning of the network, that is, ensure that no internal channel name remains after the application of the consequence rule, except for a conjunct \( \Pi_{\text{intchan}}^=A \) (which clearly cannot be removed by means of the consequence rule).
16. (3) Use the abstraction rule to get rid of the conjunct $\text{min} \chi_{\Lambda}$, and to obtain a formula for $\text{cobegin} S_1 || S_2 \text{ coend}$.

4. **Conclusion**

We introduced a formal proof system for dynamic networks of processes, and touched upon its soundness. Future work will consider the completeness of the system.
References

"A proof System for Communicating Sequential Processes"

[CH] Chen, Z.C. and Hoare, C.A.R.
"Partial Correctness of Communicating Sequential Processes."
2nd International Conference on Distributed Computer Systems,

[L] Levin, G.M.
"A Proof Technique fo Communicating Sequential Processes",

[MC] Misra, J. and Chandy, K.M.
"Proofs of Networks of Processes",

[SD] Soundararajan,N. and Dahl, O.J.
"Partial Correctness Semantics of Communicating Sequential Processes"
Research Report, Institute of Informatics, University of Oslo.