vakgroep informatica R.U. Utrecht

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

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6   3584 CD Utrecht
Corr. adres: Postbus 80.012   3508 TA Utrecht
Telefoon 030-531454
The Netherlands

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:

Proceedings of the 2nd Workshop on Logics of Programs,
D. Kozen & E. Clarke (eds.), Lecture Notes in Computer
Science, Springer Verlag, Heidelberg, 1983.

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 | skip | b | D?x | D!s |
$S_1;S_2$ | [S_1 □ S_2] | cobegin N coend |
$P(E_{in-1},\ldots,E_{in-k};E_{out-1},\ldots,E_{out-l};u_1,\ldots,u_m)$

Networks:
N ::= $S_1||S_2$

Procedure declarations:
T ::= $P(D_{in-1},\ldots,D_{in-k};D_{out-1},\ldots,D_{out-l};x_1,\ldots,x_m)$ begin S end

Programs:
R ::= $T_1,\ldots,T_n$: begin S end

The intuitive meaning of x:=s, skip and $S_1;S_2$ should be clear. [S_1 □ 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 "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 if b then S_1 else S_2 fi can be expressed in our language as [b;S_1 □ ¬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 twwo 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 cobegin - coend construct, which "hides" internal channels, i.e. no internal channel of $S_1||S_2$ is visible anymore in the process cobegin $S_1||S_2$ coend.

So in $S \equiv$ cobegin S_1 || cobegin S_2 || S_3 coend coend

with $S_1 \equiv$ D?x, $S_2 \equiv$ D!0, $S_3 \equiv$ D?y,

the S_2 process communicates with S_3 along the D channel internal to $S_2||S_3$, and not with S_1. The D channel of S_1 is an external input channel of S. We note that when no "clashes" arise between the external channel names of processes S_1, S_2 and S_3, then the semantic operator for parallel composition is associative for the network consisting of S_1, S_2 and S_3 executing concurrently, so we can write

cobegin S_1 || S_2 || S_3 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 begin - end) are assumed to be 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".

```
Q(in;out;)
begin
      shrink := false;
      while¬ shrink
         do
            [ in?val → cobegin
                                P(in,int1;out,int2;val)
                            ||
                                Q(int2;int1;)
                         coend
            □ out!"*" → shrink := true
            ]
         od
  end,
P(lin,rin;lout,rout;ownval)
begin
      shrink := false;
      while¬ shrink
         do
            [  lin?newval → largest := max(ownval,newval);
                             ownval := min(ownval,newval);
                             rout!largest
            □ lout!ownval → rin?ownval;
                             shrink := (ownval="*")
            ]
         od
  end :
begin
       cobegin  Q(D;E;)  || Userproc(E;D;)  coend
  end
```

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.

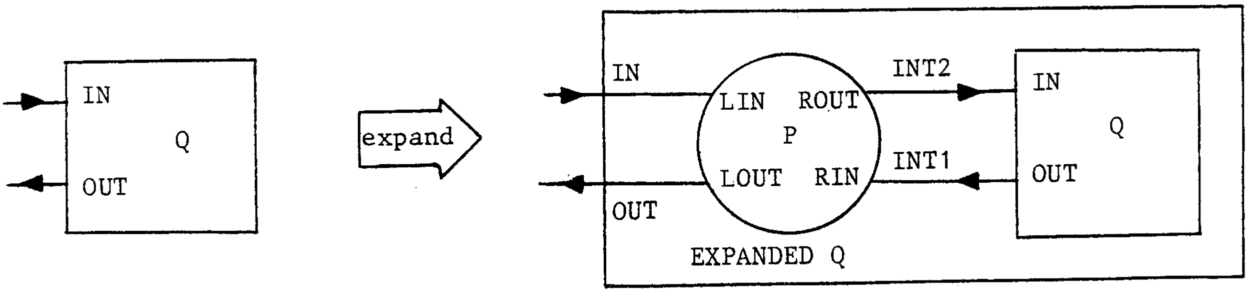

fig.1 Expansion, corresponding to the cobegin - coend part of a "Q" process.

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....

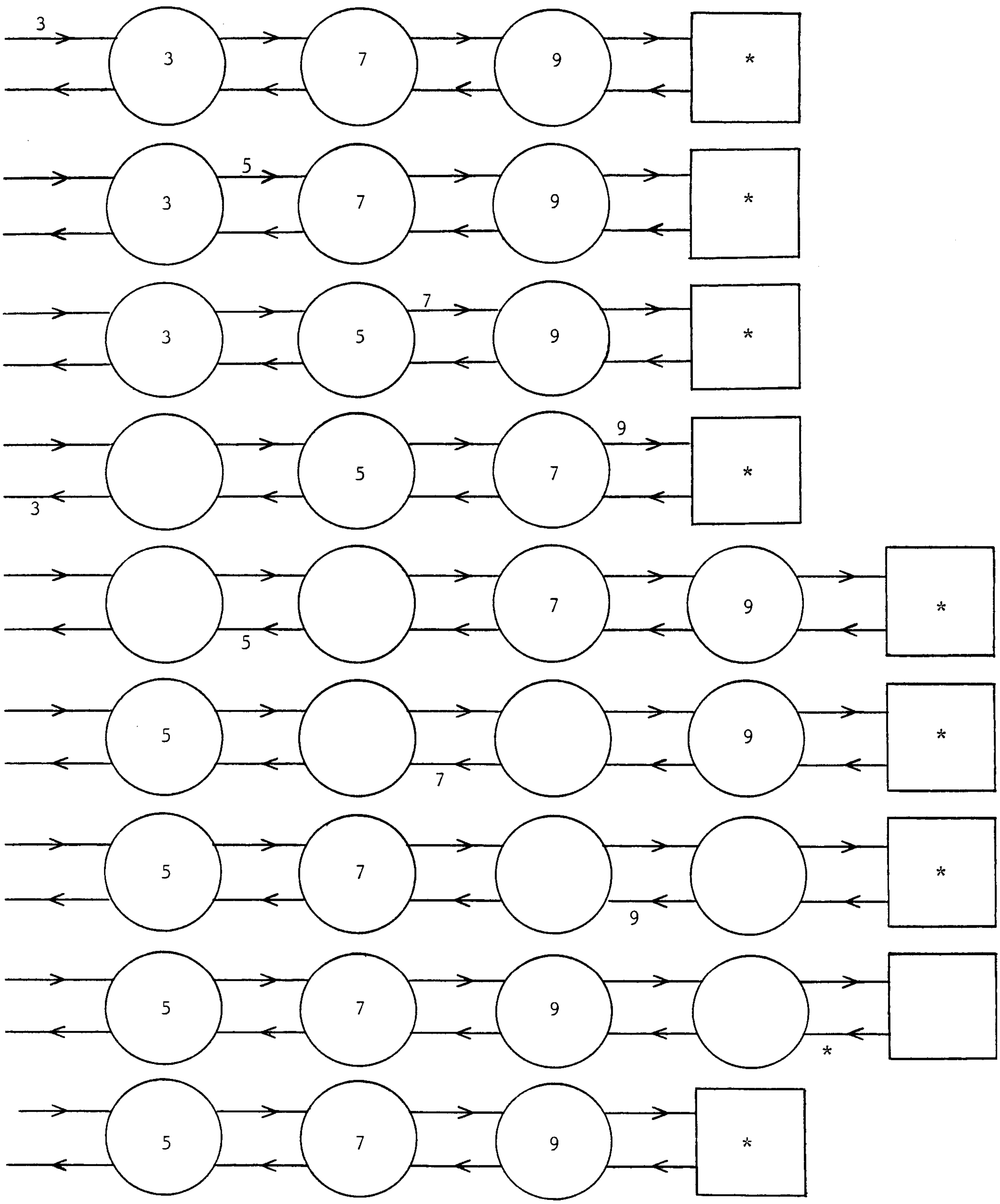

fig.2 A queue holding the values 3,7 and 9 is sent a value 5. This new value is inserted, causing the queue to expand. Before this process of insertion has come to an end, the queue is already asked for its least value. This causes a deletion of the "front" value (3); this deletion then "ripples" towards 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:

$$\frac{I_1 : \{p_1\}\ S_1\ \{q_1\}\ ,\ I_2 : \{p_2\}\ S_2\ \{q_2\}}{I_1\ \&\ I_2 : \{p_1\ \&\ p_2\}\ S_1\ ||\ S_2\ \{q_1\ \&\ q_2\}}$$

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:

$$\frac{\begin{array}{c} I' : \{p'\}\ S_1\ ||\ S_2\ \{q'\}\ \ ,\ \ I' \supset I \\ (p\ \&\ I\ \&\ \Pi_{intchan}=\Lambda) \supset (p'\ \&\ I')\ \ ,\ \ q' \supset q \end{array}}{I : \{p\}\ \underline{cobegin}\ S_1\ ||\ S_2\ \underline{coend}\ \{q\}}$$

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 $S_1||S_2$.

Λ 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:

$$\begin{array}{l} S_1 \equiv \underline{do}\ IN?x\ ;\ \underline{if}\ x>0\ \underline{then}\ D!x\ \underline{else}\ D!-x\ \underline{fi}\ \underline{od} \\ S_2 \equiv \underline{do}\ D?y\ ;\ OUT!(entier(\surd y))\ \underline{od} \\ S \equiv \underline{cobegin}\ S_1\ ||\ S_2\ \underline{coend} \end{array}$$

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
- $trace_1 \leq trace_2$: that $trace_1$ is some initial prefix of $trace_2$
- f(trace) for some function f: the trace formed by applying f componentwise

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

$$OUT = entier(\surd abs(IN)) : \{IN = OUT = \Lambda\}\ S\ \{true\}$$

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

$$D = abs(IN) : \{IN = OUT = \Lambda\}\ S_1\ \{true\}$$

and

$$nonneg(D) \supset OUT = entier(\surd(D)) : \{D = OUT = \Lambda\}\ S_2\ \{true\}$$

(where 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" $OUT = entier(\surd(D))$ which is preserved if and only if the *assumption* 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:

```
S1 ≡ x:=0 ; do D!(x+1) ; E?x od
S2 ≡ do D?y ; E!(y+1) ; OUT!y od
S  ≡ cobegin S1 || S2 coend
```

and try to prove:

$$OUT \leq (1,3,5,....) : \{OUT = \Lambda\}\ S\ \{true\}$$

from specifications for S_1 and S_2.
(OUT $\leq$ (1,3,5....) can be written more formally as:
$\forall j\ [\ 0 \leq j \leq |OUT| \supset OUT[j] = 2j-1\ ]$)
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 $C_2 \equiv E \leq (2,4,6...)\ \&\ OUT \leq (1,3,5...)$
which depends on the assumption $A_2 \equiv D \leq (1,3,5...)$

Unfortunately, the specifications:

$$A_1 \supset C_1\ :\ \{D = E = \Lambda\}\ S_1\ \{true\}$$
$$A_2 \supset C_2\ :\ \{D = E = OUT = \Lambda\}\ S_2\ \{true\}$$

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

$$C_1 \supset A_2 \supset C_2 \supset A_1 \supset C_1 \supset \cdots\cdots$$

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

$$D[1] = 0\ \&\ \forall i\ [(1<i\leq|D|\ \&\ E[i-1] = 2i-2) \supset D[i] = 2i-1\ ]\ :\ \{D{=}E{=}\Lambda\}\ S_1\ \{true\}$$

$$\forall i\ [(1\leq i\leq|OUT|\ \&\ D[i] = 2i-1) \supset (E[i] = 2i\ \&\ OUT[i] = 2i-1)]:\{D{=}E{=}OUT{=}\Lambda\}\ S_2\ \{true\}$$
$$\{D = E = OUT = \Lambda\}\ S_2\ \{true\}$$

From the conjunction of the two invariants we can, by inductive reasoning, derive an invariant of $S_1||S_2$ from which the desired invariant of S can be deduced with the abstraction rule; the "circular" reasoning is turned into a "spiral". We can paraphrase the specification for, say S_1, as follows:
Let precondition p1 $\equiv$ {D = E = Λ} 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, inspired 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:

$$\frac{\begin{array}{ll} (A_1,C_1) : \{p_1\}\ S_1\ \{q_1\}\ ,\ (A_2,C_2) : \{p_2\}\ S_2\ \{q_2\} & (1) \\ A_{net}\ \&\ C_1 \supset A_2 \quad , \quad A_{net}\ \&\ C_2 \supset A_1 & (2) \end{array}}{(A_{net}, C_1\ \&\ C_2) : \{p_1\ \&\ p_2\}\ S_1 \ ||\ S_2\ \{q_1\ \&\ q_2\}}$$

with the restrictions:

$$free(A_1,C_1,p_1,q_1) \subseteq free(S_1)$$
$$free(A_2,C_2,p_2,q_2) \subseteq free(S_2)$$

(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:

$(A_1,C_1) : \{D = E = \Lambda\}\ S_1\ \{true\}$
$(A_2,C_2) : \{D = E = OUT = \Lambda\}\ S_2\ \{true\}$
$(true,C_1\ \&\ C_2) : \{D = E = OUT = \Lambda\}\ S_1\ ||\ S_2\ \{true\}$ (*)
$(true,C) : \{OUT = \Lambda\}$ cobegin $S_1\ ||\ S_2$ coend $\{true\}$ (**)

with:
$A_1 \equiv E \leq (2,4,6...)$
$A_2 \equiv C_1 \equiv D \leq (1,3,5...)$
$C_2 \equiv A_1\ \&\ OUT \leq (1,3,5...)$
$C \equiv OUT \leq (1,3,5...)$

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_{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_{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_{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_1andC_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_{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 postconditions, 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 assignment axiom are incorporated in our system. Notice that if "S" is some noncommunicating statement, then validity of

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

boils down to: if p & 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 system, 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 particular language, we have the following axiom schemes:

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

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

$\{p\}$ skip $\{p\}$ (skip)

$\{p\}\ b\ \{p\ \&\ b\}$ (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:

$$\frac{(A,C) : \{p \,\&\, C\}\ S\ \{q\}}{(A,C) : \{p\}\ S\ \{q\}}$$

Now, what about "classical" Hoare rules like the sequential composition rule? Here we introduce proof rule schemes like for example:

$$\frac{\{p_1\}\ S_1\ \{q_1\}\ ,\ \{p_2\}\ S_2\ \{q_2\}}{\{p_3\}\ S_3\ \{q_3\}}$$

which is valid if and only if

$$\frac{(A,C) : \{p_1\}\ S_1\ \{q_1\}\ ,\ (A,C) : \{p_2\}\ S_2\ \{q_2\}}{(A,C) : \{p_3\}\ S_3\ \{q_3\}}$$

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

(notice that the same (A,C) pair is used in all formulae of the premisses as well as of the conclusion of the rule!)
We now claim that the following scheme is sound in our system:

$$\frac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}} \qquad \text{(sequential composition)}$$

(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 premisse 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 premisse 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:

$$\frac{\{p\}\ S_1\ \{q\}\ ,\ \{p\}\ S_2\ \{q\}}{\{p\}\ [\ S_1 \ \square\ S_2\ ]\ \{q\}} \qquad \text{(choice)}$$

We continue with rules for input and output commands. In these rules, we use substitutions in assertions like: A[D<v>/D] denoting the assertion formed from A by replacing terms Π_{cset}, with D ∈ cset, by Π_{cset}<D,v>. As usual, care must be taken that the variable v 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 <D,v> 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 <v> is also used for such a term.)

$$\frac{\begin{array}{c} C \ \& \ p \supset \forall v:(\ C[D\langle v\rangle/D]\ ) \\ C \ \& \ p \supset \forall v:(\ A[D\langle v\rangle/D] \supset q[D\langle v\rangle/D, v/x]\ ) \end{array}}{(A,C) : \{p\}\ D?x\ \{q\}} \qquad \text{(input)}$$

$$\frac{\begin{array}{c} C \ \& \ p \supset C[D\langle s\rangle/D] \\ C \ \& \ p \supset (\ A[D\langle s\rangle/D] \supset q[D\langle s\rangle/D]\ ) \end{array}}{(A,C) : \{p\}\ D!s\ \{q\}} \qquad \text{(output)}$$

The first premisse 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 premisse 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 premisse 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:

$S_1 \equiv [\ D!0\ ;\ E!0\ ;\ x:=0\ \square\ E!0\ ;\ D!0\ ;\ x:=1\ ]$

$S_2 \equiv E?y\ ;\ D?y$

$S \equiv$ cobegin $S_1 \ ||\ S_2$ coend

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

from: $(A_1,\text{true}) : \{\Pi_{D,E} = \Lambda\}\ S_1\ \{x=1\}$

and: $(\text{true},C_1) : \{\Pi_{D,E} = \Lambda\}\ S_2\ \{\text{true}\}$

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

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

$(A_1,\text{true}) : \{\Pi_{D,E} = \Lambda\}\ D!0\ ;\ E!0\ ;\ x:=0\ \{x=1\}$

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

$(A_1,\text{true}) : \{\Pi_{D,E} = \Lambda\}\ D!0\ \{\text{false}\}$

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.

$\overline{E}/\overline{D}$ denotes the substitution of the E-variables for corresponding D-variables.
ω denotes some special value, used to initialize all local variables of a procedure body.
var(X) denotes the free program variables of X.
chan(X) denotes the free channel variables of X.

The rules are:

$$\frac{\begin{array}{l}\langle\ \mathrm{Decl}\ \mid\ \{p\}\ P(\overline{D}_{in};\overline{D}_{out};\overline{x})\ \{q\}\ \rangle\ \vdash \qquad \text{(recursion)}\\ \langle\ \mathrm{Decl}\ \mid\ \{p\ \&\ \overline{y}=\overline{\omega}\}\ S_0\ \{q\}\end{array}}{\langle\ \mathrm{Decl}\quad \{P(\overline{D}_{in};\overline{D}_{out};\overline{x})\}\ \underline{\mathrm{begin}}\ S_0\ \underline{\mathrm{end}}\ \mid\ \{p\}\ P(\overline{D}_{in};\overline{D}_{out};\overline{x})\ \{q\}\ \rangle}$$

Provided that $\mathrm{var}(p,q) \subseteq \{\overline{x}\}$, 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.

$$\frac{(A,C)\ :\ \{p\}\ P(\overline{D}_{in};\overline{D}_{out};\overline{x})\ \{q\}}{(A[\circledcirc],C[\bullet])\ :\ \{p[\bullet]\}\ P(\overline{E}_{in};\overline{E}_{out};\overline{u})\ \{q[\circledcirc]\}} \qquad \text{(parameter substitution)}$$

where $[\circledcirc] \equiv [\overline{E}_{in}/\overline{D}_{in},\overline{E}_{out}/\overline{D}_{out},\overline{u}/\overline{x}]$

And provided that:
$(\overline{E}_{in} \cup \overline{E}_{out}) \cap \mathrm{chan}(A,C,p,q) \subseteq (\overline{D}_{in}\ \overline{D}_{out})$

$\overline{u} \cap \mathrm{var}(p,q) \subseteq \overline{x}$

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.

$$(A,C)\ :\ \{p\}\ S\ \{p\} \qquad\qquad \text{(invariance)}$$

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

$$\frac{(A_1,C_1) : \{p_1\}\ S\ \{q_1\}\ ,\ (A_2,C_2) : \{p_2\}\ S\ \{q_2\}}{(A_1\ \&\ A_2,C_1\ \&\ C_2) : \{p_1\ \&\ p_2\}\ S\ \{q_1\ \&\ q_2\}}\qquad \text{(conjunction)}$$

$$\frac{\{p\}\ S\ \{q\}}{\{p[e/f]\}\ S\ \{q[e/f]\}}\qquad \text{(freeze variable substitution I)}$$

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

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

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.

$$\frac{\begin{array}{c}(A_1,C_1) : \{p_1\}\ S_1\ \{q_1\}\ ,\ (A_2,C_2) : \{p_2\}\ S_2\ \{q_2\}\\ A_{net}\ \&\ C_1 \supset A_2\ ,\ A_{net}\ \&\ C_2 \supset A_1\end{array}}{(A_{net},C_1\ \&\ C_2) : \{p_1\ \&\ p_2\}\ S_1\ ||\ S_2\ \{q_1\ \&\ q_2\}}\qquad \text{(network)}$$

With the restrictions:

$free(A_1,C_1,p_1,q_1) \subseteq free(S_1)$
$free(A_2,C_2,p_2,q_2) \subseteq free(S_2)$

$$\frac{(A,C) : \{p\ \&\ \Pi_{intchan}=\Lambda\}\ S_1\ ||\ S_2\ \{q\}}{(A,C) : \{p\}\ \underline{\text{cobegin}}\ S_1\ ||\ S_2\ \underline{\text{coend}}\ \{q\}}\qquad \text{(abstraction)}$$

Where intchan denotes the internal channels of $S_1||S_2$, and provided that $chan(A,C,p,q) \cap intchan = \emptyset$

$$\frac{\begin{array}{c}(A',C') : \{p'\}\ X\ \{q'\}\quad \text{(with } X \equiv S \text{ or } X\equiv N\text{)}\\ A \supset A'\ ,\ C' \supset C\ ,\ p \supset p'\ ,\ q' \supset q\ ,\ p\ \&\ C \supset C'\end{array}}{(A,C) : \{p\}\ X\ \{q\}}$$

The intended usage of these rules is as follows:

(1) Use the network rule to derive a formula for $S_1||S_2$.

(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 $\Pi_{intchan}=\Lambda$ (which clearly cannot be removed by means of the consequence rule).

(3) Use the abstraction rule to get rid of the conjunct $\Pi_{intchan}=\Lambda$, and to obtain a formula for cobegin $S_1||S_2$ 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

[AFR] Apt, K.R., Francez,N. and de Roever, W.P.
"A proof System for Communicating Sequential Processes"
TOPLAS 2,3. July 1980, pp. 359-385.

[CH] Chen, Z.C. and Hoare, C.A.R.
"Partial Correctness of Communicating Sequential Processes."
2nd International Conference on Distributed Computer Systems,
IEEE 1981, 1-12.

[L] Levin, G.M.
"A Proof Technique fo Communicating Sequential Processes",
TR 79-401, Computer Science Department, Cornell University, Ithaca, New York
14853, 1979.

[MC] Misra, J. and Chandy, K.M.
"Proofs of Networks of Processes",
IEEE Transactions on Software Engineering, July 1981, pp. 417-426.

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

C49650-Y