Local Nondeterminism in Asynchronously Communicating Processes

F.S. de Boer and M. van Hulst

Utrecht University, Dept. of Comp. Sc.,
P.O. Box 80089, 3508 TB Utrecht, The Netherlands

Abstract. In this paper we present a simple compositional Hoare logic for reasoning about the correctness of a certain class of distributed systems. We consider distributed systems composed of processes which interact asynchronously via unbounded FIFO buffers. The simplicity of the proof system is due to the restriction to local nondeterminism in the description of the sequential processes of a system. To illustrate the usefulness of the proof system we use PVS (Prototype Verification System, see [ORS92]) to prove in a compositional manner the correctness of a heartbeat algorithm for computing the topology of a network.

1 Introduction

In [dBvH94] we have shown that a certain class of distributed systems composed of processes which communicate asynchronously via (unbounded) FIFO buffers, can be proved correct using a simple compositional proof system based on Hoare-logic. The class of systems introduced in [dBvH94] is characterized by the restriction to deterministic control structures in the description of the local sequential processes. An additional feature is the introduction of input statements as tests in the choice and iterative constructs. Such input statements involve a test on the contents of the particular buffer under consideration. Even in the context of deterministic sequential control structures this feature gives rise to *global* nondeterminism, because the choices involving tests on the contents of a buffer depend on the environment.

To reason about the above-mentioned class of distributed systems a buffer is represented in the logic by an input variable which records the sequence of values read from the buffer and by an output variable which records the sequence of values sent to the buffer. The communication pattern of a system then can be described in terms of these input/output variables by means of a global invariant. This should be contrasted with logics which formalize reasoning about distributed systems in terms of histories ([OG76, AFdR80, ZdRvEB85, Pan88, HdR86]). The difference between input/output variables and histories is that in the former information of the relative ordering of communication events on different buffers is lost. In [Fra92] these input/output variables are used in a

non-compositional proof system based on a *cooperation test* along the lines of [AFdR80] for FIFO buffered communication in general. A compositional proof system based on input/output variables is given in [dBvH94] for the class of systems composed of deterministic processes as described above. However, the proof system in [dBvH94] allows only a decomposition of the pre/postcondition part of the specification of a distributed system. The global invariant, which is needed for completeness and which describes the ongoing communication behaviour of the system in terms of the input/output variables, does not allow a decomposition into local invariants corresponding to the components of the system. This is due to the global non-determinism inherent in the distributed systems considered in [dBvH94].

In this paper, we investigate *local* nondeterminism, that is, we restrict to distributed systems composed of processes which may test only their own private program variables. The resulting computational model is still applicable to a wide range of applications: For example, it can be applied to the description of socalled heartbeat algorithms like, for instance, the distributed leader election problem and the network topology determination problem. The latter problem we will discuss in some detail in this paper.

We show that when restricting to local non-determinism, a complete specification of a distributed system can be derived from *local* specifications of its components, that is, from specifications which only refer to the program variables and the input/output variables of the component specified. This additional compositional feature is very important because it allows for the construction of a *library* of specified components which can be reused in any parallel context. The proof system in [dBvH94] does not allow this because part of a local specification is the global invariant which specifies the overall communication behaviour of the entire system. Moreover, the relevance of a compositional reasoning pattern [dB94, dBHdR, dBvH95, HdR86] with respect to the complexity of (mechanically supported) correctness proofs of concurrent systems lies in the fact that the verification of the local components of a system can in most practical cases be mechanized fully (or at least to a very large extent). What remains is a proof that the conjunction of the specifications of the components implies the desired specification of the entire system. This latter proof in general involves purely mathematical reasoning about the underlying datastructures and does not involve any reasoning about the flow of control. This abstraction from the flow of control allows for a greater control of the complexity of correctness proofs.

We will illustrate the above observation by proving the correctness of a heartbeat algorithm for computing the network topology using the Prototype Verification System (PVS). As the formalization of the local reasoning is straightforward, our verification effort concentrates on the second, global part of the correctness problem, viz. the proof that the conjunction of the specifications of the components implies the desired specification of the entire system.

The specification language of PVS is a strongly typed, higher-order logic. Speci-

fications can be structured into a hierarchy of parameterized theories. There are a number of built-in theories (e.g. reals, lists, sets, ordering relations, etc.) and a mechanism for automatically generating theories for abstract datatypes. Due to its high expressivity, the specification language can be invoked in many domains of interest whilst maintaining readable (i.e. not overly constructive) specifications. At the core of PVS is an interactive proof checker with, for instance, induction rules, automatic rewriting, and decision procedures for arithmetic. Moreover, PVS proof steps can be combined into proof strategies.

The reason to choose PVS is a pragmatic one: it allows a quick start, and, more importantly, its powerful engine allows one to disregard many of the trivial but tedious details in a proof, a virtue that is not shared by most of the currently available proof checkers/theorem provers. Much effort has already been invested in developing a useful tool for (automated) verification by means of PVS [CS95, Raj94].

The rest of this paper is organized as follows: In section 2, the programming language is defined. Section 3 explains the algorithm for computing the topology of a network. Then, in section 4, the proof system is introduced and its formal justification is briefly touched upon. The theorem prover PVS and the specification of the correctness of the algorithm in PVS are discussed in section 5. Finally, section 6 contains some concluding remarks and observations.

2 The programming language

In this section, we define the syntax of the programming language. The language describes the behaviour of asynchronously communicating sequential processes. Processes interact only via communication channels which are implemented by (unbounded) FIFO-buffers. A process can send a value along a channel or it can input a value from a channel. The value sent will be appended to the buffer, whereas reading a value from a buffer consists of retrieving its first element. Thus the values will be read in the order in which they have been sent. A process will be suspended when it tries to read a value from an empty buffer. Since buffers are assumed to be unbounded, sending values can always take place.

We assume given a set of program variables Var, with typical elements $x, y, \ldots$. Channels are denoted by $c, d, \ldots$. We abstract from any typing information.

Definition 1. The syntax of a statement S which describes the behaviour of a sequential process, is defined by

$$
\begin{array}{rl}
S ::= & \mathsf{skip} \\
| & x := e \\
| & c??x \mid c!!e \\
| & S_1; S_2 \\
| & []_i[b_i \to S_i] \\
| & \star[]_i[b_i \to S_i]
\end{array}
$$

In the above definition skip denotes the 'empty' statement. Assigning the value of e to the variable x is described by the statement $x := e$. Sending a value of an expression e along channel c is described by $c!!e$, whereas storing a value read from a channel c in a variable x is described by $c??x$. The execution of $c??x$ is suspended in case the corresponding buffer is empty. Furthermore we have the usual sequential control structures of sequential composition, guarded command and iterated guarded command (b denotes a boolean expression). In the example below, we only have need for simple guarded statements, which we will denote by if b then S_1 else S_2 fi and while b do S od.

In [dBvH94] we considered deterministic choice and iteration constructs which use input statements as tests. For example, the execution of a (conditional input) statement if $c??x$ then S_1 else S_2 fi consists of reading a value from channel c, in case its corresponding buffer is non-empty, storing it in x and proceeding subsequently with S_1. In case the buffer is empty control moves on to S_2. These constructs will in general enhance the capability of a deterministic process to respond to an indeterminate environment and in this respect they give rise to global nondeterminism in the sense that the choices of a process depend on the environment. Note that this is not the case in our present language, where processes can only inspect their local variables. Nevertheless many interesting algorithms described in the literature can be expressed in a programming language based on local nondeterminism. As an example we consider in the next section the algorithm for computing a network topology.

Definition 2. A parallel program P is of the form $[S_1 \parallel ... \parallel S_n]$, where we assume the following restrictions: the statements S_i do not share program variables, channels are unidirectional and connect exactly one sender and one receiver.

3 An example: Computing the network topology

We consider a symmetric and distributive algorithm for computing a network topology, which is described in [And91]. We are given a network of processes which are connected by bi-directional communication links, and each link is represented by two (unidirectional) channels, i.e. between any two processes S_i and S_j there is a channel from S_i to S_j iff there is a channel from S_j to S_i. Each

process can communicate only with its neighbors and knows only about the links to its neighbors. We assume that the network is connected. A symmetric distributed solution to the network topology problem can be obtained as follows: Each process first sends to its neighbors the information about its own links and then each of its neighbors is asked for its links. After having obtained this information each process will know its links and those of its neighbors. This it will know about the topology within two links of itself. Assuming that we know the diameter D of the network, that is, the largest distance between two nodes, iterating the above D times will solve the problem.

To formalize the above algorithm we represent the network topology by a matrix $top[1 : n, 1 : n]$ of BOOL, where n is the number of processes. $top[i, j]$ indicates whether there exists a link from process i to process j. Since we have bi-directional links we have for all processes i and j $top[i, j] = top[j, i]$. For each pair of linked processes i and j we have channels c_{ij} and c_{ji}. With respect to channel c_{ij} process i is the sender and j the receiver. The contents of each channel c is described by two variables $c??$ and $c!!$. The first variable $c??$ is local to the receiver and records all values that have been read; the second variable $c!!$ is local to the sender and records the sequence of values that were sent. Thus the input/output variables of process i are $c_{ji}??$ and $c_{ij}!!$, for all processes j such that i and j are linked. Processes communicate by sending and receiving their local views of the global topology. Each process has a local variable $lview_i$, which represents its (local) knowledge of the global topology top. Initially, $lview_i$ is intialized to the neighbors of process i, that is $lview_i[k, l] = true$ if and only if $k = i$ and $top[i, l] = true$. A local view received by a process i from one of its neighbors is stored in a local variable $nview_i$. These local views are combined by an *or*-operation on matrices, denoted by $\vee$, which is an obvious extension of the corresponding boolean operation on the truth values. The diameter of the network is given by D. The behaviour of process i is then described by the following statement:

$S_i \equiv r_i := 0;$
  while $r_i < D$
  do $j := 1;$
    while $j \leq n$
    do if $top[i, j]$
      then $c_{ij}!!lview_i$
      fi;
      $j := j + 1$
    od;
    $j := 1;$
    while $j \leq n$
    do if $top[i, j]$
      then $c_{ji}??nview_i;$
        $lview_i := lview_i \vee nview_i$
      fi;

$j := j + 1$
od;
$r := r_i + 1$
od

For a network of n processes the program for computing the network topology, i.e. the matrix *top*, is defined by $[S_1 \parallel ... \parallel S_n]$.

4 The proof system

In this section we provide a proof system for proving partial correctness and deadlock freedom of programs. To this end, we introduce correctness formulae $\{p\}P\{q\}$ which we interpret as follows:

> Any computation starting in a state which satisfies p does not deadlock, and moreover, if its execution terminates, then q holds in the final state.

Note that this interpretation is stronger than the usual partial correctness interpretation in which absence of deadlock is not required. The precondition p and postcondition q are formulae in some first-order logic. We omit the formal definition of this logic which is rather standard; here we only mention that p and q will contain besides the program variables of P the input/output variables $c??$ and $c!!$, where c is a channel occurring in P. These variables $c??$ and $c!!$ are intended to denote the sequences of values received along channel c and those sent along channel c, respectively. Logically they are simply interpreted as (finite) sequences of values (thus we assume in the logic operations like append, tail, the length of a sequence etc.).

To derive the correctness of a program P compositionally, we introduce local correctness formulae of the form $I : \{p\}S\{q\}$, where p and q are (first-order logic) assertions, allowed to refer to the variables of S only. The set of variables of a statement S consists of its program variables and those input/output variables $c??$ ($c!!$) for which c is an input channel of S (c is an output channel of S). The assertions p and q are called the precondition and postcondition, respectively, while the assertion I is called the invariant. The invariant I is a conjunction of implications of the form $Rc \rightarrow p$, where Rc denotes a predicate which indicates that the next execution step involves a read on channel c. An assertion $Rc \rightarrow p$ thus specifies that if control is about to execute a read on the channel c then p holds. The information in I will be used in the analysis of deadlock. Intuitively the meaning of a correctness formula $I : \{p\}S\{q\}$ can be rendered as follows:

> The invariant I holds in every state of a computation of S starting in a state which satisfies p and upon termination q is guaranteed to hold.

Note that the invariance of $I \equiv Rc_1 \to p_1 \wedge ... \wedge Rc_k \to p_k$ amounts to the fact that whenever control is at an input $c_i?x$, $1 \leq i \leq k$, p_i is guaranteed to hold. In other words, I expresses certain invariant properties which hold whenever an input statement (specified by I) is about to be executed. It is important to note that thus the predicates Rc_i are a kind of 'abstract' location predicates, in the sense that they refer not just to a particular location of a statement but to a *set* of locations.

Now we present the axioms and rules of our proof system.

The axiom for the assignment statement is as usual, apart from the addition of an arbitrary invariant; this is allowed because there is no communication, so none of the Rc will hold during execution of the statement.

Axiom 1 *(assignment)* $I : \{p[e/x]\}x := e\{p\}$

The output statement $c!!e$ is modeled as an assignment to the corresponding output variable $c!!$ which consists of appending the value sent to the sequence $c!!$. The operation of 'append' is denoted by '$\cdot$'. With respect to the invariant, a similar remark holds as for the assignment axiom.

Axiom 2 *(output)* $I : \{p[c!! \cdot e/c!!]\}c!!e\{p\}$

An input statement $c??x$ is modeled as a (multiple) assignment to the variable x and the input variable $c??$. The associated invariant states that when reading on c, the substituted postcondition should hold.

Axiom 3 *(input)* $Rc \to \forall v.\, p[v/x, c?? \cdot v/c??] : \{\forall v.\, p[v/x, c?? \cdot v/c??]\}c??x\{p\}$

We now give the rule for sequential composition; the rules for the choice and while statement can be obtained by extending in a similar way the usual rules for these constructs.

Rule 1 *(sequential composition)*

$$\frac{I : \{p\}S_1\{q\},\ I : \{q\}S_2\{r\}}{I : \{p\}S_1; S_2\{r\}}$$

So in order to prove that I is an invariant of $S_1; S_2$ one has, naturally, to prove that I is both an invariant of S_1 and S_2.

We have the following local consequence rule:

Rule 2 *(local consequence)*

$$\frac{I' \to I,\ p \to p',\ I' : \{p'\}S\{q'\},\ q' \to q}{I : \{p\}S\{q\}}$$

We introduce the expression c as an abbreviation of the expression $c!! - c??$. By $c!! - c??$ we denote the suffix of the sequence $c!!$ (i.e. the sequence of values sent) which is determined by its prefix $c??$ (i.e. the sequence of values read). Thus c represents the contents of the buffer, that is, the values sent but not yet read. The empty sequence we denote by ϵ.

In preparation of the parallel composition rule, we first observe that a possible deadlock configuration of a program P is characterized by: Every process is either done or about to execute a read on a channel for which the corresponding buffer is empty; moreover at least one process is not yet done. Suppose $P = [S_1 \parallel \ldots \parallel S_n]$ and each S_i has input channels $c_1^i, \ldots, c_{m_i}^i$. Hence we have the predicates $Rc_1^i, \ldots, Rc_{m_i}^i$ for each $i \in \{1, \ldots, n\}$. Furthermore assume a postcondition q_i for each of the S_i. Now we introduce a set of assertions $C(P)$, the disjunction of which characterizes all possible deadlock configurations of P:

$$C(P) = \{\textstyle\bigwedge_i p_i \mid p_i \equiv Rc_k^i \wedge c_k^i = \epsilon, \text{ for some } k \leq m_i, \text{ or } p_i \equiv q_i, \\ \text{and there exists } j : p_j \not\equiv q_j\}.$$

Note that each assertion $p \in C(P)$ characterizes a *set* of possible deadlock configurations.

Definition 3. Given some local postconditions $q_1, \ldots, q_n$, we define for local invariants $I_1, \ldots, I_n$ the assertion $DF(I_1, \ldots, I_n)$ as

$$\bigwedge_{p \in C(P)} \left(\bigwedge_{i=1}^{n} I_i \wedge p \to \textbf{false} \right)$$

The above assertion $DF(I_1, \ldots, I_n)$ expresses that the conjunction of the local invariants is inconsistent with any possible deadlock configuration, i.e. the assertion $\bigwedge_{i=1}^n I_i$ guarantees deadlock freedom.

Local correctness formulas then can be combined into correctness formulas of an entire program as follows:

Rule 3 *(parallel composition)*

$$\frac{I_i : \{p_i\}S_i\{q_i\} (i = 1, \ldots, n), DF(I_1, \ldots, I_n)}{\{\textstyle\bigwedge_i p_i\}[S_1 \parallel \ldots \parallel S_n]\{\bigwedge_i q_i\}}$$

In the premise of the above rule the formula $DF(I_1, ..., I_n)$ is implicitly assumed to be defined with respect to the local postconditions $q_1, \ldots, q_n$. The compositional method of proving deadlock freedom incorporated in the above rule can be best understood by comparing it with the standard way of proving deadlock freedom using the *proof outlines*. For example in [AFdR80], given proof outlines of the components of a CSP program $P \equiv [S_1 \parallel \ldots \parallel S_n]$, absence of deadlock can be proved by first determining statically all possible deadlock configurations. Such a configuration consists of a n-tuple of local locations (one location for each component). Each possible deadlock configuration then is characterized by the conjunction of the assertions associated with its locations by the given proof outlines. Absence of deadlock then can be established by showing that the assertion associated with each possible deadlock configuration is equivalent to false. The main difference with our deadlock analysis lies in the use of the predicates Rc which do not refer to a specific location but represent a set of locations, namely all those locations where the corresponding process is about to execute a read on channel c. In our case then deadlock freedom can be established by showing that the conjunction of the local invariants, which provide information about the local states of processes when these are about to execute a read, is inconsistent with any possible deadlock configuration. This abstraction from specific locations, which is due to the restriction to local nondeterminism, allows for the simple compositional proof rule for parallel composition described above.

Apart from the above rule for parallel composition we also have the usual consequence rule for programs. With respect to reasoning about global states we moreover have for each channel c the following axiom of asynchronous communication:

$$c?? \leq c!!$$

where $\leq$ denotes the prefix ordering on sequences.

The formal justification of the proof system, i.e. soundness and (relative) completeness can be proved in a rather straightforward manner using a compositional semantics which associates with each statement S a meaning

$$\mathcal{M}(S) \in \Sigma \to \mathcal{P}(\Sigma \times Chan \to \mathcal{P}(\Sigma))$$

(Σ denotes the set of states, a state being a function which assigns values to the program variables and the input/output variables, and *Chan* denotes the set of channel names). Here $\langle \sigma', f \rangle \in \mathcal{M}(S)(\sigma)$, with $f \in Chan \to \mathcal{P}(\Sigma)$, indicates that σ' is the result of a terminating computation of S starting from σ, and every intermediate state σ'' just before an input on a channel c belongs to $f(c)$. In other words, $f(c)$ collects all the intermediate states which occur just before an input on channel c is executed. Formally we then define for $I \equiv \bigwedge_i Rc_i \to p_i$,

> $\models I : \{p\}S\{q\}$ iff for every pair of states σ and σ' and function $f \in Chan \to \mathcal{P}(\Sigma)$, such that $\langle \sigma', f \rangle \in \mathcal{M}(S)(\sigma)$ and p holds in σ, it is the case that q holds in σ' and p_i holds in every state $\sigma'' \in f(c_i)$.

The semantics of a program can be defined in terms of the meaning $\mathcal{M}(S)$ of its components by a straightforward 'translation' of the parallel composition rule of the proof system. Moreover it is rather straightforward to prove the correctness of the compositional semantics with respect to an operational semantics. More details can be found in the technical report [dBvH96].

5 Automated verification in PVS

In this section, we will show how the network topology determination algorithm can be specified and verified using PVS.

The specification to be proved is

$$\{\bigwedge_i (lview_i[i,l] = top[i,l] \wedge (j \neq i \rightarrow lview_i[j,l] = \mathit{false}))\}$$
$$[S_1 \parallel ... \parallel S_n]$$
$$\{\bigwedge_i lview_i = top\}$$

In words, if initially for every i, $lview_i$ is initialized to the neighbours of i, then the program $[S_1 \parallel ... \parallel S_n]$ terminates in a state in which for any i, $lview_i$ equals the actual network topology top .

Using the local proof rules, it is not difficult to derive the following local specification for each S_i (it is implicitly assumed that the indices j and k range over the neighbours of i):

$$\bigwedge_j Rc_{ji} \rightarrow (\bigwedge_k |c_{ik}!!| = r_i \wedge \bigwedge_{k<j} |c_{ki}??| = r_i \wedge \bigwedge_{k>j} |c_{ki}??| = r_i - 1):$$
$$\{lview_i[i,l] = top[i,l] \wedge (j \neq i \rightarrow lview_i[j,l] = \mathit{false})\}$$
$$S_i$$
$$\{q_i \wedge \bigwedge_j |c_{ij}!!| = |c_{ji}??| = D\}$$

For the moment, we do not consider yet the first part of the postcondition q_i, which we will consider in detail later in this section. The invariant informally states that when a process is ready to receive on channel c_{ji}, all its outgoing channels have length r_i, as well as its in-going channels from processes with index smaller than j, and the in-going channels from all processes from index j upward have length $r_i - 1$.

To derive the specification for $[S_1 \parallel ... \parallel S_n]$ we have to show first that the condition for deadlock freedom holds, so that we can apply the parallel composition rule. Then there remains to show that the conjunction of the q_i implies the global postcondition $\bigwedge_i lview_i = top$.

As to the first problem, we have to show for any $p \in C(P)$: $\bigwedge_i I_i \wedge p \rightarrow \mathit{false}$. The proof of this is far from trivial, and omitted for reasons of space. Essentially,

it involves starting at some process waiting for an input, and tracking down the processes on which it is waiting until arriving at the first process again or at a terminated process, which in both cases leads to a contradiction. The intricacy of the proof stems from the fact that the processes may run 'out of phase' to a considerable degree.

In the rest of this section, we will focus on the second essential part of the proof, which involves an application of the global consequence rule. We now focus on the specification of this problem in PVS.

Specifications in PVS are organized in theories, which may depend on other theories via an importing mechanism. In particular, any theory may import from the set of built-in theories. As an example of this, in the theory `processes` below the type `nat` is (silently) imported. Theories may be parameterized, as in our case: the parameter `n` denotes the number of processes that participate in the algorithm. The first axiom below takes care that we are dealing with at least 2 processes. The type `process` is defined as a subtype of the natural numbers, i.e. the primitive type `nat`. The type `pairset` will be used further on in the definition of type `links`; it fixes the type of sets of 2-tuples of processes.

```
processes [ n: nat ] : THEORY
BEGIN

process   : TYPE = {m: nat | 1 <= m AND m <= n}

pairset   : TYPE = setof[[process,process]]
```

The variable declarations which follow below should be self-explanatory. The constraints on the type `links` express the properties that any network topology should possess: no channel should connect a process with itself (`nonrefl`), channels are bidirectional (more accurately: the existence of a channel implies the existence of the reverse channel) (`symmetric`) and any process should be connected to at least one process (`connected`) (we provide the definition of `nonrefl` only). The projection functions `proj_1` and `proj_2` are built-in accessor functions on tuples.

```
m,m1,k     : VAR nat

i,j,i1,j1,i2,j2  : VAR process

z, z1      : VAR [process,process]
```

```
p          : VAR pairset


nonrefl    : pred[pairset] =
             LAMBDA (p):
                (FORALL(z):
                   (member(z, p)) IMPLIES proj_1(z) /= proj_2(z) )


links      : TYPE = { p: pairset | nonrefl(p) AND
                                   symmetric(p) AND
                                   connected(p)    }


l          : VAR links
```

The following fragment should be self-explanatory.

```
%
% neighbors(l,i) yields the set of neighbors of process i in
% linkset l
%

neighbors: [links,process -> setof[process]] =
           LAMBDA (l,i): { j | EXISTS (z): member (z,l) AND
                               proj_1(z) = i AND proj_2(z) = j }


%
% path(l,i,j,m) = TRUE iff there exists a path of length m
% between i and j in linkset l
%

path    : pred[[links,process,process,nat]] =
          LAMBDA (l,i,j,m):
            (EXISTS(sp: sequence[process]):
                      i = sp(0) AND j = sp(m) AND
                      (FORALL (m0: nat): m0 < m IMPLIES
                      (member( sp(m0 + 1), neighbors(l,sp(m0)))) ))
```

The next two lemmas are useful in proving the larger lemmas below. Their proof in PVS requires minimal effort, while they provide more clarity in bigger proofs. `chain` states that if there exists a path from i to j of length $m+1$ then there exists a neighbor of i which has distance m to j.

```
chain    : LEMMA
             FORALL (m:nat):
               (path(l,i,j,m+1)
                 IMPLIES
                   (EXISTS (j1:process): member(j1, neighbors(l,i))
                    AND path(l,j1,j,m) ))
```

```
zeropath : LEMMA
             path(l,i,j,0)
               IMPLIES
                 i = j
```

The type `matrix` is used as representation for the data objects in our domain, viz. $lview_i$ and $nview_i$ in the algorithm. Each channel c_{ij} is described by the channel variables `inchan(i,j)` for $c_{ij}??$ and `outchan(i,j)` for $c_{ij}!!$.

```
matrix   : TYPE = [process,process -> bool]

index    : TYPE = {m:nat | m < n-1}

ix,ix2   : VAR index

chan     : TYPE = [[process,process],index -> matrix]

inchan   : chan

outchan  : chan
```

`topold(l,i)` yields the matrix with only the `i`-th row filled in according to the neighbor set of `i` with respect to `l`. Thus it corresponds to the value of $lview_i$ at the beginning of the algorithm.

```
topold   : [links,process -> matrix] =
```

```
LAMBDA(l,i): (LAMBDA(i1,j1):
              IF i = i1 THEN member(j1, neighbors(l,i))
              ELSE FALSE
              ENDIF )
```

Using the rules of the proof system for local correctness formulas it is straightforward to derive the following postcondition, for each i (note that any free variable is implicitly universally quantified over, so that `postcond` below expresses the conjunction over all i). Note that, because the postcondition directly relates the values of indexed channel variables (which are matrices), there is no need to introduce local variables. The postcondition, referred to as q_i above, is plainly expressed by

$$c_{ij}!![ix] = \left(\texttt{topold(l,i)} \ \vee \bigvee_{\substack{i2 \in \texttt{neighbors(l,i)} \\ 0 \leq ix2 < ix}} c_{i2,i}??[ix2]\right)$$

In words, the matrix that is sent out to any j in the ix-th (outer) loop equals the original topology of the sender, or-ed with all inputs from its neighbors so far (note that $\vee$ denotes the logical *or* lifted to matrices). Wrapping together all postconditions, this amounts to the following PVS expression:

```
postcond : AXIOM
           member(j,neighbors(l,i)) IMPLIES
             outchan((i,j),ix) =
                        (LAMBDA(i1,j1):(topold(l,i)(i1,j1) OR
                            (EXISTS(i2:process):
                             (EXISTS(ix2:index):
                              (member(i2,neighbors(l,i)) AND
                              ix2 < ix AND
                              inchan((i2,i),ix2)(i1,j1)))) ))
```

The next lemma `chansplit` which is used in the proof of `main` below was proven with induction on `k`. It expresses the following relation:

$$c_{ij}!![k+1] = \left(c_{ij}!![k] \ \vee \bigvee_{j2 \in \texttt{neighbors(l,i)}} c_{j2\,i}??[k]\right)$$

It reduces the matrix that has been sent over c_{ij} in the $k+1$-th (outer) loop to an expression consisting of matrices that were sent and received by i in the k-th loop.

```
chansplit: LEMMA
           forall(k):
           k < n-2
             IMPLIES
             (member(j,neighbors(l,i))
              IMPLIES
                outchan((i,j),k+1)(i1,j1) =
                (outchan((i,j),k)(i1,j1) OR
                 (EXISTS(j2): member(j2,neighbors(l,i))
                   AND inchan((j2,i),k)(i1,j1) )))
```

Before coming to the main theorem, we show a few other helpful lemmas:

```
%
% lessdist is true iff there is a path between i and j with length
% smaller than or equal to k
%

lessdist : [links,process,process,nat -> bool] =
           LAMBDA(l,i,j,m):
            EXISTS(m1):(m1 <= m AND path(l,i,j,m1))


nextneigh : LEMMA
            (lessdist(l,i,j,m+1) AND i /=j )
               IMPLIES
                  (EXISTS(i2):(member(i2,neighbors(l,i))
                         AND  lessdist(l,i2,j,m)))


ldist1    : LEMMA
            lessdist(l,i,j,m) IMPLIES lessdist(l,i,j,m+1)


ldist2    : LEMMA
            (NOT lessdist(l,i,j,m+1))
```

```
        IMPLIES
           FORALL(j1): (member(j1,neighbors(l,i))
                        IMPLIES
                         (NOT lessdist(l,j1,j,m)))
```

We now come to the main theorem which states that the k-th output over channel c_{ij} is a matrix that equals `topold(l,i1)` with respect to row `i1` if the distance in the network between `i` and `i1` is less than or equal to k, and otherwise it yields FALSE on that row. In particular, it follows from this theorem (again using local reasoning) that after D executions of the loop, the value of $lview_i$ corresponds with the network topology *top*. The second conjunct may not seem too exciting, but is needed to keep the induction going.

```
main    : THEOREM
          k < n-1 IMPLIES
           ((lessdist(l,i,i1,k)
              IMPLIES
               FORALL (j): member(j, neighbors(l,i))
                   IMPLIES
                 (outchan((i,j),k)(i1,j1) = topold(l,i1)(i1,j1)))
            AND
            ((NOT lessdist(l,i,i1,k))
              IMPLIES
               FORALL (j): member(j, neighbors(l,i))
                   IMPLIES
                 (outchan((i,j),k)(i1,j1) = FALSE)) )


END processes
```

The proof of `main` is currently about 15 pages. Possibly this can be improved by defining some clever strategies (in fact macros of proof steps). Perhaps more interesting is to construct as general as possible a proof, so that it can be re-used in the light of small changes.

6 Conclusions

We have shown how the restriction to local nondeterminism gives rise to a simple compositional proof system based on Hoare logic for distributed systems composed of processes which interact asynchronously via unbounded FIFO buffers.

We used the theorem prover PVS in a non trivial application of the proof system to the correctness of a heartbeat algorithm for computing the topology of a network.

In general we believe that a fruitful line of research with respect to automated verification is the syntactic identification of classes of distributed systems which allow a simple compositional reasoning pattern.

References

[AFdR80] K.R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM-TOPLAS*, 2(3):359–385, 1980.

[And91] Gregory R. Andrews. *Concurrent Programming, Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[CS95] D. A. Cyrluk and M. K. Srivas. Theorem proving: Not an esoteric diversion, but the unifying framework for industrial verification. In *IEEE International Conference on Computer Design (ICCD) '95*, Austin, Texas, October 1995.

[dB94] F.S. de Boer. Compositionality and completeness of the inductive assertion method for concurrent systems. In *Proc. IFIP Working Conference on Programming Concepts, Methods and Calculi*, San Miniato, Italy, 1994.

[dBHdR] F.S. de Boer, J. Hooman, and W.-P. de Roever. *State-based proof theory of concurrency: from noncompositional to compositional methods.* Draft of a book.

[dBvH94] F.S. de Boer and M. van Hulst. A proof system for asynchronously communicating deterministic processes. In B. Rovan I. Prívara and P. Ružička, editors, *Proc. MFCS '94*, volume 841 of *Lecture Notes in Computer Science*, pages 256–265. Springer-Verlag, 1994.

[dBvH95] F.S. de Boer and M. van Hulst. A compositional proof system for asynchronously communicating processes. In *Proceedings MPC'95*, Kloster Irsee, Germany, 1995.

[dBvH96] F.S. de Boer and M. van Hulst. Local nondeterminism in asynchronously communicating processes. Technical report, Utrecht University, 1996. In Preparation.

[Fra92] N. Francez. *Program Verification.* Addison Wesley, 1992.

[HdR86] J. Hooman and W.-P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current trends in concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 343–395. Springer-Verlag, 1986.

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

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[Pan88] P.K. Pandya. *Compositional Verification of Distributed Programs.* PhD thesis, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, INDIA, 1988.

[Raj94] S. Rajan. Transformations in high-level synthesis: Formal specification and efficient mechanical verification. Technical Report CSL-94-10, CSL, 1994.

[ZdRvEB85] J. Zwiers, W.-P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In *Proc. ICALP'85*, volume 194 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.