Defining a Verification Methodology for Distributed Algorithms

Tereza C. M. B. Carvalho 1* Ana R. Cavalli 2 Sidney Monreal Martin 1*
EPUSP/BRISA INT HUMANA Informática
1 University of Sao Paulo, Av. Prof. Luciano Gualberto, Tr 3, N 158, CEP 05508 Cidade Universitaria, Sao Paulo - SP, BRAZIL
2 Institut National des Telecommunications, Les Epinettes, 9 rue Charles Fourier, 91011 EVRY Cedex, FRANCE

Abstract: This paper presents a verification methodology based on simulation for the design and verification of distributed algorithms. The proposed methodology is illustrated by the verification of a fault-tolerant traversal algorithm that solved the naming problem of a network. Errors have been found and after correction, a new version of the algorithm is given.

Resumo: Este artigo apresenta uma metodologia de verificação baseada em si
mulaçao para o projeto e verificaçao de algortimos distribuidos. A metodolo
gia proposta e ilustrada atraves da verificaçao de um algoritmo tolerante a
falhas que resolve o problema de atribuiçao de nomes de nos de uma rede. Er
ros encontrados foram corrigidos e uma nova versao do algoritmo e mostrada.

1. Introduction

The specification languages for protocols and distributed algorithms: ESTELLE, SDL and LOTOS have been standardized by ISO and CCITT in the recent years. They represent the methodological and scientific progress accomplished in relation to the informal languages used before.

These languages, that have been originally conceived for the description of communication protocols and services of the OSI (Open Systems Interconnection) architecture, can be easily adapted to the description of any distributed algorithm.

Many reasons make these languages the ideal tools for the description of protocols and distributed systems: simplicity of concepts, expression power, precise semantic, stable definition, controlled evolution, large community of users and reduction of cost for software development.

Different verification techniques have been developed for these languages. We have studied and contributed to the development of these techniques [CAV87] [CAV88] and [CAR91]. In this last work a fault tolerant distributed algorithm is described using the SDL

* The research was supported by the CNPq (Brazil) and the INT (France)

language and simulated using a prototyping tool. The simulations permitted the detection of errors and, after correction, the verification of the algorithm.

The present work extends [CAR91] in the following way: we consider a "multiformalism" point of view , i. e., developing verification techniques common to all three languages. More precisely, we want to extend the use of any of the specification languages as a support for the design and verification of distributed algorithms.

On the other hand, it is well known that no safety techniques exist for the verification of the correctness of distributed algorithms. This paper proposes a methodology based on the use of simulation for the design and verification of the correct behavior of distributed algorithms. Its main purpose is to increase the confidence in the correctness of algorithms whose mathematical proofs are hard to obtain.

Our work is related to [JAR88], where protocols are described using the language ESTELLE and verified by VEDA a verification-oriented simulation tool. Even if in this work we use the same specification language, our point of view is different, we propose the use of any of the specification languages for the description of the algorithms and we don't suggest any specific tool. All verification tools that can perform the steps that we propose in our methodology will be useful in accomplishing our goals.

In this paper, we illustrate our proposal giving the specification and verification of a fault tolerant traversal algorithm described in [BEA90]. The algorithm is described using the specification language ESTELLE, and simulation is performed using EDT (ESTELLE Development Tool Set), a tool developed at BULL and at the present time at INT [BUD91].

The paper is organized as follows: Section 2 gives a brief presentation of ESTELLE and EDT; Section 3 presents an outline of the methodology; Section 4 illustrates its use on a fault tolerant algorithm and Section 5 gives the conclusions of this work. Finally, Appendix A gives the commented ESTELLE specification of the module responsible for the creation of the simulation environment and Appendix B gives the ESTELLE specification of the algorithm with the modifications that have been done to correct errors found in the original version of the algorithm.

2. The Formal Description Language ESTELLE and its Toolset

This section presents a brief description of the specification language ESTELLE and EDT (ESTELLE Development Tool Set). For a complete description of the language see [ISO89] and for EDT see [BUD91].

ESTELLE is an extension of the Pascal language based on an extended state transition model, i. e., a model of a nondeterministic communicating finite state machine. A system in ESTELLE can be seen as a hierarchical structure of communicating finite state machines, running in parallel, communicating by exchange of messages and by sharing, in a restricted way, some variables.

Each communicating component is in fact an instance of a module defined within the ESTELLE specification. The behavior of a module and its internal structure are specified respectively by a set of transitions that it may perform and by the definition of its children (submodules) together with their interconnections. Modules may exchange messages via interaction points. The received messages are appended to an unbounded FIFO queue associated to each interaction point. Communication is asynchronous.

EDT (ESTELLE Development Tool Set) is a set of integrated tools developed at BULL S.A. and further extended at INT. EDT includes an ESTELLE compiler [BUL90A], which translates an ESTELLE specification (after static errors detection) into C language source codes and an ESTELLE Simulator/Debugger [BUL90B], which allows the validation of a specification with respect to dynamic errors.

The user of ESTELLE Simulator/Debugger can define a simulation scenario, consisting of simple and macro commands, which may include the description of properties/anomalies to be detected during simulation. This last description can be defined as being an observer of the behavior of the system .

Both global and local properties can be analyzed by means of an interactive or automatic (predefined scenario and properties to be detected) simulation. The global properties give insight, for instance, of deadlocks and undesired sequences of transitions. Local properties concern errors that occur while executing a transition or evaluating a transition's enabling conditions; for instance, uninitialized variables and lost outputs.

The simulations can be extended with "time constraints" defined by the user. These time constraints may reflect the known execution speeds of components of a real computer architecture in which the specification is to be implemented.

3. Outline of the Methodology

This section presents an outline of the major steps of the methodology that we are proposing. It consists basically of the use of simulation to verify the correct behaviour of distributed and parallel algorithms. And its main purpose, as said before, is to increase the confidence in the correctness of algorithms whose mathematical proofs are hard to obtain.

First of all, it is necessary to create an environment for the simulation of the algorithm based on the formal description language, e.g., ESTELLE, SDL and LOTOS, on the corresponding toolset to be used and on the algorithm itself. At this level, the algorithm is seen from a functional point of view, i.e., we are not interested in internal details of it.

The creation of this environment implies the definition of a system architecture comprising the modules responsible for the implementation of the algorithm to be verified, the auxiliary modules necessary to test the algorithm, the interactions among the defined modules and the observation points used to analyze the simulation carried out and to verify the execution of the algorithm.

After the system architecture is defined, the following step consists of the specification of the algorithm and the test environment using the adopted formal description language. In this phase, all the eventual ambiguities of the informal description of the algorithm must be eliminated. It is worth noting that this new formal description becomes the actual description of the algorithm, replacing the original one.

Finally, after the system is completely described in the adopted formal description language, simulation sessions are performed. To allow a gradual verification and analysis of the algorithm, the simulations are conceived in two ways: **User Controlled** and **Random Simulations.**

The **User Controlled Simulation** has the purpose of exercising the algorithm in user predefined conditions to test at least once all the decision branches of the algorithm. This approach intends to verify if the algorithm works out in simple situations. This type of simulation is supposed to require more participation of the algorithm designers or verifiers in order to plan the specific situations for the algorithm verification and to analyze the obtained results.

The **Random Controlled Simulation** is performed in order to test the algorithm in a more exhaustive manner. In this case the algorithm is simulated a great number of times in randomly determined conditions that are expected to create complex situations. As this simulation type normally is supposed to take long runs, it is planned to be carried out unattended with the results being monitored and analyzed automatically to detect inconsistencies. If inconsistencies are found, the algorithm designer or verifier must analyze the corresponding simulation sessions and, eventually, make the necessary corrections on the algorithm. It is important to emphasize that each time the algorithm is modified the two types of simulations must be run again to detect undesirable collateral effects.

When the system passes all the simulations without errors, the description of the algorithm is considered reliable. This description is then extracted from the system architecture and becomes the ultimate description of the algorithm.

4. An Illustration of the Methodology

This section illustrates the use of the proposed methodology, i.e., the use of simulation to verify the correct behaviour of distributed algorithms, taking as example a fault-tolerant traversal algorithm that solves the naming problem of a graph as proposed in [BEA90].

4.1. Case Study: A Fault-tolerant Traversal Algorithm

The algorithm that we have verified through the proposed methodology performs the traversal of a graph and solves the naming problem, giving a distinct identifier (a number) to each of its nodes. The traversal is done in a faulty environment, i.e., one where nodes can crash.

In this algorithm, each node of the graph is described as being composed of two logical layers: Application and Communication layers. The Application layer is responsible for the naming part of the algorithm and is not sensitive to node failures. On the other hand, the Communication layer is responsible for the traversal of the graph and for the handling of definitive failures of graph nodes (crash). The Communication and Application layers communicate with each other through a set of predefined messages.

In the absence of node failures, a token with an identification field (a number) performs a depth-first traversal of the graph, starting at an initiator node. Each time this token visits a new node, the Communication layer sends it to the Application layer. This layer then updates the token identification field (increments it) and uses it to name the current node. The token is returned back to the Communication layer to continue the traversal.

In case of node failures, a token can be lost. Because of this, another token has to be generated in order to complete the traversal. The communication layer is responsible for this new token generation when it detects a crash failure in its current son node (the last not already visited neighbour node to whom a token has been sent). For this purpose, the Communication layer always keeps a copy of the token, which is updated each time the token is sent to a new son.

As a consequence of the token's creation it is possible, at a given time, to have more than one token circulating through the graph. It must then be decided which token will continue the traversal, i.e., it is necessary to have a method to compare tokens. This is achieved using additional token fields, which have information concerning the token history (a time stamp), and a metric for comparison. In this way, when a Communication layer receives a token, it compares it with the copy previously saved. If the received token is worse, it is discarded. Otherwise, it continues the traversal.

A complete description of this algorithm can be found in [BEA90]. For the purpose of the present paper, we used a later version of this algorithm derived in [CAR91] that corrects some problems found in the original version. The final version of the algorithm is given in Appendix B.

4.2. System Architecture for Simulation

In this phase of the methodology, it is necessary to create an environment for the simulation of the algorithm based on the formal description language, on the corresponding toolset to be used and on the algorithm itself.

In the case of the traversal algorithm, we have adopted ESTELLE as the formal description language and EDT as a debugger / simulator tool [BUL90B], which were briefly described in Section 2.

According to the description of the algorithm, the system to be modelled is a directed graph, representing a set of interconnected nodes, with a defined topology. Each node of the graph can be described in ESTELLE as a pair of modules called "na" and "nc". These modules are responsible, respectively, for the Application and Communication layers specified in the traversal algorithm. The graph is then modelled by several instances of this pair of modules created by a system module called "graph". To accomplish this, the "graph" module must have an internal data structure describing the graph topology. The asynchronous parallelism of the algorithm execution among the graph nodes are adequately modelled by the nondeterminism implied by the **activity** attribute given to the modules "na" and "nc". This system architecture is depicted in Figure 4.1.

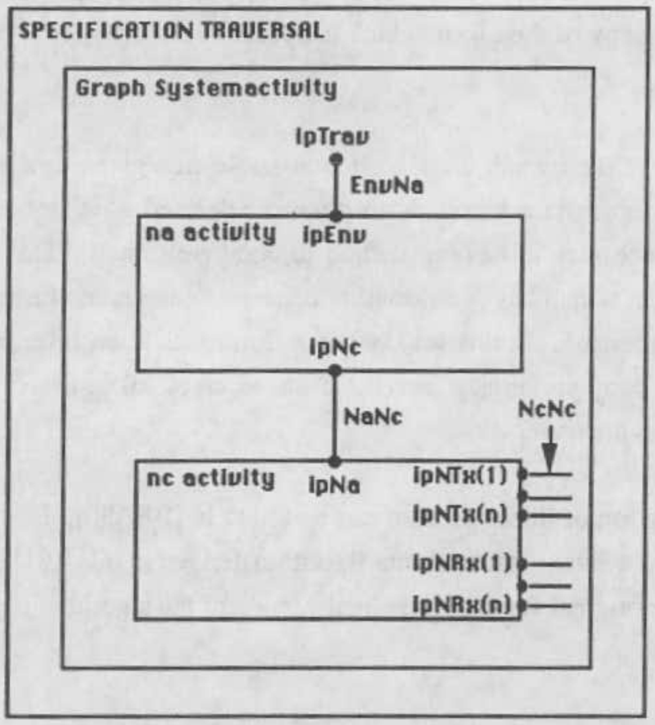

Figure 4.1 - System Architecture

In this architecture, the "na" module of the initiator node, i.e., the node that is going to initiate the traversal algorithm, is connected to the "graph" module through the channel **EnvNa.** Through this channel, the "na" module receives the message *NameGraph* used to initiate the traversal algorithm.

The "na" and "nc" modules of each node are in turn interconnected through the channel called **NaNc.** Through this channel are exchanged four types of messages. The first one, *TravInit(info)*, is sent by the "na" module, after the reception of the *NameGraph* message, to the "nc" module in order to begin the traversal algorithm. The "nc" module that receives such message is then called "initiator". The second message, *TravEnd(info)*, is sent by the "nc initiator" to the "na" module to inform that the traversal algorithm was already executed. These two messages carry a parameter *info* with the number used by the Application layer to identify the nodes. The third message, *TravVisit(t)*, is sent by the "nc" module to the "na" module to inform that the "nc" module has received a new token. The last message, *TravCont(t)*, is the response to the *TravVisit* message. It is issued by the "na" module to the "nc" module, after processing the *info* field of the token *t*. The "nc" module uses then the token *t* received in the *TravCont(t)* message to continue the traversal. The parameter *t* of these two messages is constituted of the following fields: *info*, which contains the information processed by the Application layer; *numSites*, which indicates the number of nodes already visited by this token; *stampSize*, which indicates the size of the *stamp* field of this token; and *stamp*, which is a vector with a sequence of integers representing a time stamp.

Finally, the "nc" modules of the neighbour nodes are interconnected through the channel called NcNc. Through this channel are exchanged two types of messages: *TokenMsg* and *Crash*. The *TokenMsg(t)* message contains the token that traverses the graph. The parameter *t* is the same as described above for the *TravVisit* and *TravCont* messages. The *Crash* message acts as a hardware indication of a permanent failure of a node in a real system. It is issued by a "nc" module to inform its neighbours that it has crashed (the algorithm assumes that the neighbours of each crashed node are warned of its failure).

The conditions related to how and when a node crashes are not defined in the original description of the algorithm. Therefore, it is necessary to include these definitions as part of the environment required to carry out the simulations. In this way, in order to get more meaningful simulations, it was decided that the nodes would signalize failures through the *Crash* message only after the reception of a token. Upon the token reception a node to crash can then behave in two different ways: consuming or not consuming the received token. In the first case, the *Crash* messages are sent without processing the token, i.e., the token is lost. In the latter case, the node processes the token, sends it to the corresponding neighbour and then sends the *Crash* messages. In both cases all further messages received by the crashed node are ignored.

The description of the behaviour (not failing; failing consuming the token; failing not consuming the token) of each node is stored in a data structure of the "graph" module. Each node in turn will be aware of its behaviour when it is created by the "graph" module.

In order to allow the analysis and verification of the algorithm execution, observation points were specified corresponding to data structures of the "graph" and "na" modules, which are examined and displayed by observers defined in the script language of the used toolset, EDT. The observed data structure of the "graph" module describes the graph topology, the initiator node of the traversal algorithm and the nodes to crash. In the case of the "na" module, the observed data structure contains information related to the identification of the node achieved by the algorithm execution.

4.3. System Description

After the System Architecture is defined, the following step consists of specifying the algorithm and the test environment using the adopted formal description language, ESTELLE. As an illustration, the Appendix A presents the ESTELLE description of the "graph" module, which is responsible for the creation of the test environment.

As it can be seen in this appendix, the behaviour of the "graph" module is modelled through a finite state machine with the following states: **Initial, Active** and **EndSimul.**

In the transition from the **Initial** state to the **Active** state, it is created the graph upon which the traversal algorithm will be executed. This is accomplished through the creation of different instances of the "na" and "nc" modules and the interconnection of the corresponding interaction points, according to a data structure describing the graph topology. When the "nc" module instance is created, it is also defined the behaviour (no fail; fail consuming the token; fail not consuming the token) of each node through a specific parameter, *tpFail.* At the end of this transition the traversal algorithm is started through the message *NameGraph* sent to the "nc" module instance of the initiator node.

During the algorithm execution the "graph" module remains at the **Active** state. When this execution is finished, the "na" module of the initiator node sends a *NameEnd* to the "graph" module, causing a transition from the **Active** state to the **EndSimul** state. This transition is detected by an observer that analyses and stores the simulation results, i.e., the naming of the graph achieved in each test condition and an indication of its correctness.

The "graph" module remains at this state if it is a **User Controlled Simulation** or if it was executed the last **Random Simulation.** Otherwise, there is a transition to the state **Initial.** In this transition, the graph data structures are initialized using a random number generator to define other conditions for the next simulation.

4.4. Simulation

Finally, after the system is completely described in the adopted formal description language, we have the simulation sessions.

In the case of the traversal algorithm, considering the **User Controlled** and the **Random** simulations, a great number of simulations were carried out. The control of each simulation is done through the use of macros and observers defined in the script language of the used toolset, EDT. We use basically two types of macros. The first one specifies the graph topology, the initiator node and the nodes to crash for the **User Controlled** simulation or just the graph topology for the **Random** simulation, updating a data structure of the "graph" module. The second one is responsible for starting the simulations and activating the observers.

The results of the simulations are saved in a file to be analyzed later. If problems are found, e.g., the naming is not correctly achieved for a specific test suite, the simulation can be repeated under the same conditions that generated the problem, presenting this time a more detailed trace of the algorithm execution. This detailed trace is obtained through the use of a special observer. This trace is then further analyzed and, if some inconsistency is detected in the algorithm, its description is modified accordingly and the simulation cycles are repeated.

In the case of the traversal algorithm, these simulations enabled the detection of an error that resulted from a complex situation related to a simultaneous occurrence of events between the different layers of the same node. This situation was not predicted in the original version of the algorithm. Once the corresponding modification of the algorithm was accomplished, more than three hundred simulations were performed and it was verified that the naming of the graph was correct in all these tested situations. Thus, these results allowed us to have confidence that the final version of the algorithm is robust and stable. The ESTELLE description of this final version of the algorithm is presented in the Appendix B.

5. Conclusions

In this paper we have presented a methodology for the design and verification of distributed algorithms. Its application has been illustrated through the formal description and the verification of a fault-tolerant traversal algorithm.

The methodology was developed from a "multiformalism" point of view, i.e., it is independent of the used formal description language and of the used toolset.

Nevertheless, the studied example showed that efforts must be done in order to define an appropriate environment for the simulation. This environment must be based on the chosen formal description language, on the tool used for simulation and on the algorithm itself. The formal description of the algorithm must be done carefully to avoid changes in the original version of it.

References

[BEA90] Beauquier J., Gastin P. and Vilain V., *Toward a Methodology for Designing Fault-tolerant Algorithms*, Rapport LITP, Universite' Paris VII, September 1990.

[BUD91] Budkowski S., *Estelle Development Toolset (EDT)*, Rapport DSR, Institut National des Télécommunications, June 1991.

[BUL90A] BULL/MARBEN *Estelle to C Compiler - User Reference Manual*, BULL S.A., Direction Methodology / MDL, Rue Jean Jaures, 78430 Les Clayes s/s Bois, France, May 1990.

[BUL90B] *Estelle Simulator / Debugger (Edb) Version 2.4 - User Reference Manual*, BULL S.A., Direction Methodology / MDL, Rue Jean Jaures, 78430 Les Clayes s/s Bois, France, May 1990.

[CAR91] Carvalho T.C.M.B., Cavalli A.R. and Martin S.M., *Verifying a Fault-Tolerant Algorithm by Simulation*, Rapport DSR, Institut National des Télécommunications, May 1991.

[CAV87] Cavalli A.R. and Horn F., *Proof of Specification Properties by Using Finite-State Machines and Temporal Logic,* Proc. Seventh IFIP Symposium on Protocol Specification, Testing and Verification, Zurich, North-Holland, May 1987.

[CAV88] Cavalli A.R. and Paul E., *Exhaustive Analysis and Simulation for Distributed Systems, Both Sides of the Same Coin*, Distributed Computing, Vol 2, Springer Verlag, 1988.

[CCI88] CCITT. *SDL, Specification and Description Language*, Recommendation Z100, International Consultative Committee for Telephony and Telegraphy, Geneva, 1988.

[ISO89].ISO/IEC JTC 1/SC 21 N4230, *Proposed Draft Addendum to ISO 9074:1989 : Estelle Tutorial*, December 1989.

[JAR88] Jard C., Groz R. and Monin J.F., *Development of VEDA, a prototyping tool for distributed algorithms*, IEEE Transactions on Software Engineering, Vol 14, No 3, March 88.

Appendix A

This appendix presents the ESTELLE specification of the "graph" module, which is responsible for the creation of the simulation environment. It is an annotated specification in order to facilitate its comprehension.

```
body GraphBody for Graph;
   (************************ Declaration Part *************************)
  const
     MaxSizeStamp = 8;
     MaxNumNodes = 15;
     MaxIdentNum = 30;
     MaxNumNeighbours = 10;
     None = 0;
  type
     NodeIdx = 1..MaxNumNodes;
     CommLineNum = 1..MaxNumNeighbours;
     None_CommLineNum = None..MaxNumNeighbours;
     StampIdx = 1..MaxSizeStamp;
     StatusConnection = 0..1;           (* for the graph topology description (nbs below) *)
     StatusCommLine = (NotVisited, Visited, Crashed);
     StatusSet = array [CommLineNum] of StatusCommLine;
     Failures = (N, CT, NCT);        (* None; Consuming Token; Not Consuming Token *)
     ClearMode = (AllNeighbours, ExceptCrashed);
     TokenCmpResult = (StLower, StEqual, StHigherSubst, StHigherNotSubst);
     Token = record
              info: integer;
              numSites: integer;
              stampSize: 0..MaxSizeStamp;
              stamp: array [StampIdx] of integer;
           end;
  var       (* These variables are updated at simulation/debugging time by script files:       *)
    numNodes: NodeIdx;                        (* Number of nodes of the graph to be simulated *)
    numNbs: array [NodeIdx] of CommLineNum;  (* Number of neighbours of each node *)
    (* The next communication line number to be used for a connection to each node          *)
    nextCommLine: array [NodeIdx] of CommLineNum;
    (* Matrix specifying the connections among the nodes comprising the graph (topology): *)
    (* each edge is specified only once.                                                    *)
    nbs: array [NodeIdx,NodeIdx] of StatusConnection;
    tpFail: array [NodeIdx] of Failures;                        (* Type of failure of each node *)
    initiatorNode: NodeIdx;                              (* Initiator of the traversal algorithm *)
    (* Variables to control the simulation:                                                  *)
    tpTest: (ControlledSimul, RandomSimul);        (* Type of simulation to be carried out *)
    numRandomSimul: integer;           (* Number of random simulations to be carried out *)
    maxNumFail: integer;    (* Maximum number of failures in each random simulation         *)
    identVector: array [1..MaxIdentNum] of boolean; (* Node identification vector used for *)
                                                          (* verification of the algorithm execution *)
  (*================== Channels Definition ==================*)
  channel EnvNa (User, Provider);
    by User:
        NameGraph;
    by Provider:
        NameEnd;
  channel NaNc (User,Provider);
    by User:
        TravInit (info: integer);
        TravCont (t: Token);
    by Provider:
        TravVisit (t: Token);
        TravEnd (info: integer);
channel NcNc (Tx, Rx);
   by Tx, Rx:
        TokenMsg (t: Token);
        Crash;
(*==================== M o d u l e s   D e f i n i t i o n ====================*)
module Na activity;
   ip  ipEnv: EnvNa (Provider);
       ipNc: NaNc (User);
end;
body NaBody for Na; external;
module Nc activity (numCommLines: CommLineNum; fail: Failures);
   ip  ipNa: NaNc (Provider);
       ipNTx: array [CommLineNum] of NcNc (Tx);
       ipNRx: array [CommLineNum] of NcNc (Rx);
end;
body NcBody for Nc; external;
modvar  NaMod: array [1..MaxNumNodes] of Na;
        NcMod: array [1..MaxNumNodes] of Nc;
ip  ipTrav: EnvNa (User);          (* Internal ip to start the Traversal and to detect its end*)
(*==================== S t a t e s   D e f i n i t i o n ====================*)
state Initial, Active, EndSimul;
(*********************** i n i t i a l i z a t i o n   P a r t ***********************)
initialize
   to Initial
      var i, j: integer;
      begin
        initiatorNode := 1;
        for i := 1 to MaxNumNodes do begin
           nextCommLine[i] := 1;  (* Number of the first communication line available *)
           tpFail[i] := N;         (* Node is not going to crash *)
           for j := 1 to MaxNumNodes do
              nbs[i,j] := 0;       (* Neighbours not connected *)
        end;
        for i := 1 to MaxIdentNum do
           identVector[i] := false;   (* Nodes without identification *)
      end;
(************************** T r a n s i t i o n   P a r t ***********************)
(*==================== I n i t i a l   S t a t e   ====================*)
trans
   from Initial
      to Active
         provided true
          var i, j: integer;
          begin
            (* Create nodes of the graph *)
            for i := 1 to numNodes do begin
               init NaMod[i] with NaBody;
            end;
            for i := 1 to numNodes do begin
               init NcMod[i] with NcBody (numNbs[i], tpFail[i]);
            end;
            (* Make the connections *)
            (*   Connect the Environment and the Application layer of the Initiator node *)
            connect ipTrav to NaMod[initiatorNode].ipEnv;
            (*   Connect the Application and Communication layers of each node *)
            for i := 1 to numNodes do begin
```

```
              connect NaMod[i].ipNc to NcMod[i].ipNa;
          end;
          (*   Connect the Communication layers defining the graph topology *)
          for i := 1 to MaxNumNodes do begin
             for j := 1 to MaxNumNodes do begin
                if nbs[i,j] = 1
                   then begin
                      connect NcMod[i].ipNTx[nextCommLine[i]]
                               to NcMod[j].ipNRx[nextCommLine[j]];
                      connect NcMod[i].ipNRx[nextCommLine[i]]
                               to NcMod[j].ipNTx[nextCommLine[j]];
                      nextCommLine[i] := nextCommLine[i] + 1;
                      nextCommLine[j] := nextCommLine[j] + 1;
                   end;
             end;
          end;
          output ipTrav.NameGraph;   (* Initiate the traversal algorithm *)
       end;
(*======================== Active State =====================*)
trans
   from Active
      to EndSimul
         when ipTrav.NameEnd
            begin        (* This state exists to allow an observer to present the *)
            end;         (* result of the execution of the algorithm              *)
(*======================== EndSimul State ====================*)
trans
   from EndSimul
      to Initial
         provided (tpTest = RandomSimul) and (numRandomSimul >= 1)
            var i, rd, node: integer;
            begin        (* Prepare another test sequence (simulation) *)
               numRandomSimul := numRandomSimul - 1;
               for i := 1 to MaxIdentNum do  (* Clear the ident. vector used by the observer *)
                  identVector[i] := false;
               for i := 1 to numNodes do begin     (* Destroy node module instances *)
                  terminate NaMod[i];
                  terminate NcMod[i];
               end;
               for i := 1 to MaxNumNodes do begin  (* Update data for the next simulation *)
                  nextCommLine [i] := 1;
                  tpFail [i] := N;
               end;
               initiatorNode := ((*$C$ (int)random() %*) numNodes) + 1;
               for i := 1 to maxNumFail do begin
                  node := ((*$C$ (int)random() %*) numNodes) + 1;
                  while (node = initiatorNode) do begin
                     node := ((*$C$ (int)random() %*) numNodes) + 1;
                  end;
                  rd := (*$C$ (int)random() %*) 3;
                  case rd of
                     0: tpFail [node] := N;
                     1: tpFail [node] := CT;
                     2: tpFail [node] := NCT;
                  end;
               end;
            end;
end;
```

Appendix B

This appendix presents the ESTELLE specification derived from the simulations of the traversal algorithm that corresponds to the "nc" process description (Communication layer) . The errors found and corrected in the original algorithm are pointed out in bold face. The comments were included to facilitate the comprehension of the algorithm.

```
body NcBody for Nc;
   (*********************** Declaration Part *********************)
   var initiator: boolean;            (* Indicate if this node is the initiator of the  *)
                                      (* traversal algorithm                            *)
      nodeVisited: boolean;           (* Indicate if this node was already visited      *)
      father: None_CommLineNum;       (* Number of the 1st communication line           *)
                                      (* that sent me the newest valid token            *)
      currentSon: None_CommLineNum; (* Number of the last communication line to *)
                                      (* whom I have sent a token                       *)
      myCopy: Token;                  (* My copy of the token                           *)
      tokenN: array [CommLineNum] of Token; (* Token from the neighbour N              *)
      visitedSet: StatusSet;          (* Set of visited communication lines             *)
      potenVisitedSet: StatusSet;     (* Set of potentially visited comm. lines         *)
   state CommIdle;
   (* primitives:                                                                       *)
   (*  function cmpTokens (t1, t2: Token): TokenCmpResult;                              *)
   (*  function tstSubstEqToken (t1, t2: Token ): boolear                               *)
   (*  function searchNeighbour (v: StatusSet; numNbs CommLineNum):                     *)
   (*        None_CommLineNum;                                                          *)
   (*  procedure clearSets (m: ClearMode);                                              *)
#include "./trav.proc"
   (********************** Initialization Part *******************)
   initialize to CommIdle
   begin
     initiator := false;              (* Initialize working variables *)
     nodeVisited := false;
     father := None;
     currentSon := None;
     clearSets (AllNeighbours);       (* Clear Visited and PotentiallyVisited sets *)
   end;
   (*********************** Transition Part *********************)
   (*=================== state CommIdle =================*)
   (*------- Reception of the TravInit(Info) message from the Application layer -------*)
   trans
     from CommIdle
       to CommIdle
         when ipNa.TravInit (info)
           begin
             initiator := true;       (* I'm the initiator of the traversal *)
             nodeVisited := true;
             myCopy.info := info;     (* Initialize my copy of the token *)
             myCopy.numSites := 1;
             myCopy.stampSize := 0;
             if numCommlines > 0
               then begin             (* I have some neighbours: *)
                 currentSon := 1;     (* Take one of the neighbours *)
                 output ipNTx[currentSon].TokenMsg (myCopy);
               end
               else                   (* I don't have any neighbour: *)
                 output ipNa.TravEnd (info);
           end;
```

```
(*----------- Reception of the Traversal.Continue (t) from the Application layer -----------*)
trans
  from CommIdle
    to CommIdle
      when ipNa.TravCont (t)
        begin
          myCopy.info := t.info;
          currentSon := searchNeighbour (visitedSet, numCommLines);
          if currentSon <> None
            then                (* There is some neighbour to be visited *)
              output ipNTx[currentSon].TokenMsg (t)
            else                (* There are no more neighbours to be visited *)
              output ipNtx[father].TokenMsg (t);
        end;
(*------------------ Reception of the token T sent by the neighbour N --------------------*)
trans
  from CommIdle
    any n: CommLineNum do
      when ipNRx[n].TokenMsg (t)
        to CommIdle
          var i: CommLineNum;
          begin
            if not nodeVisited
              then begin            (* I haven't received any token so far *)
                nodeVisited := true;(* Update my status *)
                t.numSites := t.numSites + 1;
                father := n;
                myCopy := t;
                visitedSet[n] := Visited;
                output ipNa.TravVisit (t);
              end
              else begin            (* I have already received some token *)
                case cmpTokens (t,myCopy) of    (* Compare Stamps *)
                  StEqual:          (* case 1) T.Stamp =st MyCopy.Stamp *)
                    begin
                      if currentSon <> n
                        then begin (* The Token is not from my Current Son *)
                          potenVisitedSet[n] := Visited;
                          tokenN[n] := t;
                          output ipNTx[n].TokenMsg (t);
                        end
                        else begin (* The Token is from my CurrentSon *)
                          (* Add PotenVisited and CurrentSon to the Visited set *)
                          for i := 1 to numCommLines do
                            if (potenVisitedSet[i] = Visited)
                              then begin
                                visitedSet[i] := Visited;
                                potenVisitedSet[i]:=NotVisited;
                              end;
                          visitedSet[currentSon] := Visited;
                          myCopy := t;
                          currentSon := searchNeighbour(visitedSet,
                                                        numCommLines);
                          if ( currentSon <> None )
                            then  (* There are some neighbours to be visited *)
                              output ipNTx[currentSon].TokenMsg (t)
                            else  (* There are no more neighbours to visit*)
                              if initiator
                then    (* I am the Initiator *)
                   output ipNa.TravEnd (t.info)
                else    (* I am not the Initiator *)
                   output ipNTx[father].TokenMsg (t);
        end;
  end;
StLower:           (* case 2) T.Stamp <st MyCopy.Stamp        *)
  begin            (* This Token will not achieve its traversal *)
  end;             (* It is stopped here                        *)
StHigherNotSubst:  (* case 3) T.Stamp >st MyCopy.Stamp        *)
                   (* T isn't a substitute for MyCopy         *)
  begin
    t.numSites := t.numSites + 1;
    father := n;
    myCopy := t;
    clearSets (ExceptCrashed);      (* Clear PotentiallyVisited and *)
                                    (* Visited sets                 *)
    visitedSet[n] := Visited;
    currentSon := None;
    output ipNa.TravVisit (t);
  end;
StHigherSubst:   (* case 4) T is a substitute for MyCopy *)
  begin
    if currentSon <> n
      then begin         (* The Token is not from my CurrentSon *)
        potenVisitedSet[n] := Visited;
        tokenN[n] := t;
        output ipNTx[n].TokenMsg (t);
      end
      else begin         (* The Token is from my CurrentSon *)
        (* Add to Visited set: CurrentSon and PotentiallyVisited *)
         (* N such that T is a substitute for Token[N] or T is   *)
         (* equal to it                                          *)
        for i := 1 to numCommLines do begin
          if potenVisitedSet[i] = Visited
             then begin
               if tstSubstEqToken (t, tokenN[i])
               then begin
                 visitedSet[i] := Visited;
               end;
               potenVisitedSet[i] := notVisited;
             end;
        end;
        visitedSet[currentSon] := Visited;
        myCopy := t;
        currentSon := searchNeighbour (visitedSet,
                                       numCommLines);
        if currentSon <> None
          then    (* There are some neighbours to be visited *)
             output ipNTx[currentSon].TokenMsg (t)
          else    (* There are no more neighbours to be visited.*)
             if initiator
               then    (* I am the Initiator *)
                  output ipNa.TravEnd (t.info)
               else    (* I am not the Initiator *)
                  output ipNTx[father].TokenMsg (t);
        end
  end
```

```
                end;
            end;
        end;
(*------------------------- Reception of Crash from the neighbour N ----------------------*)
trans
  from CommIdle
    to CommIdle
      any n: CommLineNum do
        when ipNRx[n].Crash
          var i: CommLineNum;
          begin
            (* Remove N from the Visited and PotentiallyVisited sets *)
            visitedSet[n] := Crashed;
            potenVisitedSet[n] := Crashed;
            if n = currentSon
              then begin      (* My current son is the crashed neighbour: *)
                currentSon := searchNeighbour (visitedSet,
                                                numCommLines);
              if currentSon <> None
                then begin          (* There are some neighbours to be visited *)
                  (* Concatenate MyCopy.NumSites to MyCopy.Stamp; *)
                  myCopy.stampSize := myCopy.StampSize + 1;
                  myCopy.stamp[myCopy.StampSize] := myCopy.NumSites;
                  for i := 1 to numCommLines do      (* Potentially Visited = [ ] *)
                    if (potenVisitedSet[i] = Visited )
                      then
                        potenVisitedSet[i] := NotVisited;
                  output ipNTx[currentSon].TokenMsg (myCopy);
                end
                else begin          (* There are no more neighbours to be visited *)
                  if initiator
                    then          (* I am the Initiator *)
                      output ipNa.TravEnd (myCopy.info)
                    else begin    (* I am not the Initiator *)
                      (* Concatenate MyCopy.NumSites to MyCopy.Stamp; *)
                      myCopy.stampSize := myCopy.StampSize + 1;
                      myCopy.stamp[myCopy.StampSize] := myCopy.NumSites;
                      output ipNTx[father].TokenMsg (myCopy);
                    end;
                end;
            end;
        end;
```