Enumeration protocol in Estelle: an exercise in stepwise development

P. Dembiński
*Institute of Computer Science, Polish Academy of Sciences**
Ordona 21, 01-237 Warsaw, Poland
e-mail: piotrd@ipipan.waw.pl

Abstract

The paper sketches a design process that starts with a mathematical solution of the enumeration problem and ends up with a distributed, efficient, and successfully terminating protocol in the specification language Estelle. The subsequent versions of the protocol are obtained from their predecessors by well controlled and small modifications that (almost) obviously preserve correctness. The target, successful (successful with arbitrarily high probability) solution allows forgetting theoretical "impossibility results", which prove non-existence of such successful distributed solutions for some classes of networks.

Keywords

Distributed algorithms, Formal Specification, Verification, FDT Estelle.

1 INTRODUCTION

The enumeration problem is one of a number of similar problems (election of a leader, mutual exclusion, termination, etc.) that have been studied because of their importance for distributed systems and networks, i.e., systems composed

* Part of this work was written when the author visited INT, Evry, France

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35394-4_29

S. Budkowski et al. (eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*

of a number of independent processes running in parallel and communicating by messages. A solution to the problem (an algorithm) is considered distributed if the processes run the same algorithm, no process is privileged, and their computation decisions are solely based on local information and information gained through messages coming from neighbour processes. A distributed algorithm in the above sense is called *protocol.*

In the enumeration problem, networks of processes are abstracted by graphs that are: finite, connected, and without self-loops. Nodes correspond to processes and the graph relation represents the communication links between them. The enumeration problem consists in finding a distributed algorithm (protocol) that terminates with all processes (nodes) of a network (graph) *properly* enumerated, i.e., each process of an N-process network gets a unique label from the set {1, 2, ... , N}.

It is known that for some graphs (*ambiguous* graphs) there is no distributed enumeration algorithm that always terminates successfully (so called, *impossibility results*, e.g., in (Litovsky at al., 1993)). Several solutions have been proposed that worked for specific classes of graphs (like, e.g., rings). In (Mazurkiewicz, 1997), an elegant protocol, that works for all unambiguous graphs, is presented and proved correct. The solution is kept at a high-level of abstraction to reduce the interference of particular realization choices to a necessary minimum.

The aim of this work has been to implement the Mazurkiewicz protocol, i.e., the protocol from (Mazurkiewicz, 1997), in the specification language Estelle (ISO, 1989) and to experiment with Estelle supporting tools EDT (Estelle Development Toolset, (Budkowski, 1992)). Also, the work seemed a good case study to show that such an implementation can be obtained in a way that preserves correctness of the original solution proved at a higher, mathematical level. This occurred feasible, but it also occurred that trying to translate, as close as possible, the main protocol ideas from (Mazurkiewicz, 1997) into environment where sharing variables is not possible (a direct availability of values of internal variables of processes in the neighbourhood of a given process, is one of the simplifying assumptions in (Mazurkiewicz, 1997)), leads to an inefficient and clumsy solution. Moreover, not all properties of this solution were exactly the same as in the original. The obtained protocol is discussed in Sec. 2 with illustrative extracts from its Estelle specification (EP1: Enumeration Protocol 1).

Being not satisfied with the obtained Estelle specification, a modification of the original protocol has been proposed that resulted in a solution which is both economic and has exactly all properties of the original. The solution (EP2) is analyzed and described in Section 3. The way it is obtained from EP1 shows again that its correctness can be deduced directly from this predecessor version, hence, indirectly from its mathematical original. In the following section the notion of ambiguity of a graph is briefly discussed in order to explain non-existence of an (always successful) distributed enumeration algorithm for ambiguous graphs. A very simple extension to EP2 (EP3) makes the protocol "always" successful, i.e., even for ambiguous graphs. The word *always* is

appropriate since success can be achieved with arbitrary high probability. Again, correctness of EP3 protocol follows directly from the correctness of EP2. A full Estelle specification of EP1, EP2 and EP3 can be found in (Dembiński, 1996). The last "probabilistic" solution led the author to a radical reformulation of the protocol into a randomized one, but this reformulation is the subject of a separate paper (Dembiński, 1998).

2 MAZURKIEWICZ ALGORITHM AND ITS IMPLEMENTATION BY A NETWORK OF IDENTICAL COMMUNICATING PROCESSES.

The enumeration protocol in (Mazurkiewicz, 1997) is assumed to be performed by a network of sequential processes, each one for a node of the graph to be enumerated (named). Similarly as for the graph itself, we use the term *neighbour* (*neighbourhood*) of a process to mean another process (all processes) representing a node (all nodes) directly in the graph relation with the node of the process. Each process has its main local variables (we are using slightly different notation than in (Mazurkiewicz, 1997)):

own_name - storing the current name (number) assigned to the process (initially all processes have no names, i.e., *own_name* = *0*),

known_names - storing a set of names (numbers assigned to neighbour processes and currently known to the process; initially, *known_names* = ∅),

knowledge - storing a set of pairs (*name*, *set_of_names*) representing the current knowledge of names of other processes in the network and their *known_names* at the time the knowledge was acquired (initially, *knowledge* = ∅)

In the informal description of the algorithm in (Mazurkiewicz, 1997), the author explains the local behavior of each graph node by means of message exchange between neighbour nodes. In fact, such an exchange of messages is necessary if we really think of the algorithm realized in a network of sequential processes. However, in (Mazurkiewicz, 1997), for the sake of simplicity of the correctness proof of the enumeration protocol, an indivisible, atomic computation step of each node is realized as if each process had a direct read/write access to all local variables of its neighbourhood. The atomic step (identical for each process) repeated until the *termination_condition* is satisfied can be described, in Pascal-like terms, as follows:

repeat
  if *knowledge_in_the_neighbourhood_is_not_the_same* **then** *unify*;
  if *renaming_condition* **then** *rename*
until *termination_condition*

Without going into details of the conditions and procedures, it is worth to point out that the condition *knowledge_in_the_neighbourhood_is_not_the_same* and the procedures *unify* and *rename* use and change values of local variables of the neighbourhood processes. That is why the whole step is kept atomic. With such an approach elegant proofs are provided for the following results:

1. The protocol always terminates. If graph is not ambiguous (see Section 4) the protocol terminates with its nodes properly (or successfully) enumerated (*successful termination*). If graph is ambiguous the protocol may terminate in a state in which no process can continue (*unsuccessful termination*) but the graph nodes are not properly enumerated (see example in Section 4).

2. The protocol may reach any given proper enumeration for an arbitrary finite and connected graph, i.e., even for ambiguous graphs the protocol may reach a proper enumeration.

3. If the protocol terminates successfully, then each node knows about this termination. Moreover, each node has enough information to reconstruct the whole graph (i.e., the protocol solves the graph recognition problem).

4. The protocol solves the election problem (see, e.g. (Bougé, 1988)) for all unambiguous graphs (it chooses a leader-node and any node can be chosen as a leader).

5. The protocol is distributed in that, in principle, processes that do not have a common neighbour can run in parallel (the parallelism is modelled by interleaving). Otherwise, their actions must be mutually excluded.

The proofs in (Mazurkiewicz, 1997) follow from a key invariant property of the above algorithm. The property says that each computation step preserves so called *connectivity of the graph enumeration.* Assume that a graph has N nodes. An *enumeration* is a function from the graph nodes V into the set $\{1, 2, ..., |V|\}$. An enumeration f is called *connected* if, for every x in V such that $f(x) = n+1$, there is y in V with $f(x) = n$. The proof of invariant goes by induction. Of course, if nodes have no names (all names are set to 0), then the property holds. Each step may rename a node and therefore define a new enumeration which is increasing its maximal value. The first one is obviously connected (it gives 1 as a name to exactly one node). It is sufficient then to prove that a computation step cannot destroy this connectivity. Finally, if a node gets $|V|$ as its name, then by the connectivity argument, a proper enumeration is achieved and the algorithm terminates with each node having knowledge about the whole graph. To complete the proof it should be added that, for an unambiguous graph and for any non-terminal state, eventually a node is renamed (i.e., a new enumeration function is eventually created).

Our first goal is to specify a distributed version of the protocol, i.e., such in which local variables of a process cannot be accessed by another one. Exchange of messages can be the only way of knowing their values.

Below, we describe a distributed solution (EP1) in terms of a (well known) model assuming that each process is described by a finite set of *guarded commands*. A *guard* is a boolean condition over a space of the process *internal states* and the contents of its (unbounded) *communication mailbox* storing incoming *messages*. A *command* defines an *action* that may change an internal state, may remove some messages from the process's mailbox, and it may send messages to some other processes *connected* to the given process. A message sent to a process is (immediately) appended to its mailbox. The action defined by a process command is assumed *indivisible* (or *atomic*). The communication is *asynchronous* in that a process may always send a message or, in other words, operations of sending and receiving messages are not synchronized. A process, in each of the computation steps, performs one of the commands from among those for which their guards evaluate to *true*. The behavior of a *network of processes* is described by interleaved sequences of processes' actions.

Specifying the Mazurkiewicz protocol in the above model is fairly straightforward if one is not going into details. That is what we are trying to do in this paper: an informal description completed with some corresponding parts in Estelle to illustrate how close an Estelle specification can mimic the colloquial (though precise) language. The basic idea in this first distributed solution is to replace the direct access to the values of local variables of neighbour processes by the exchange of messages that include these values.

Enumeration Protocol (solution) 1 (EP1)

In the description below we refer to the following elements:

- messages: *request* - asking for actual contents of local variables of neighbour processes,
  response - responding to a request and containing the requested information (values of *own_name*, *known_names* and *knowledge*),
  new_knowledge - a message containing, for each neighbour, a newly created value of its *known_names* and a unified *knowledge*, for all of them; one such message is transmitted to each of the neighbours of the process that previously requested information from them.

Estelle definitions of a network node and some types of basic data stored in nodes and exchanged between them:

{ A specification of the architecture of enumeration protocol EP1 is

```
  given, for a graph of size given by 'graph_size' constant and generated
  by 'graph_creation' procedure in 'initialize' part of the specification.
  A node of the network (graph) is defined by the module header 'node'
  and the module body 'node_body' with its input and output ports
  specified by vectors of length 'graph_size' of interaction points (ip).
  Ports of the instances of nodes are connected in  part 'initialize'
  of the specification, by communication channels of type 'link(R1, R2)'
  and according to the edge structure of the graph. Messages circulate
  through channels. Allowed messages are defined within channel 'link'.
  Sets of names ('L_set') are defined as sets of integers from the set
  'V = 1 .. graph_size', while pairs composed of a name and a set of
  names have the record 'K_el' definition.
  Knowledge is represented by an array of 'max' length. Its actual
  contents are reduced to 'K_el' elements that are not equal to
  (max, empty). This not very elegant solution is chosen because of the
  Estelle restriction that does not tolerate dynamic arrays and forbids
  pointers in messages.
  The 'systemactivity' attribute of the specification means that behavioral
   semantics are given by interleaving.}

specification EP1 systemactivity;
 default common queue;
 const
  graph_size = any integer;
  max = any integer;      {greater than graph_size, eg. graph_size^2}
 type
  zero_one = 0..1;
  V = 1..graph_size;
  neighbour_representation = array[V] of zero_one;
  graph_representation = array[V] of neighbour_representation;
  L_set = set of  0..graph_size;
  K_el = record
         ng : integer;
         known: L_set;
        end;
  K_set = array[1..max] of K_el;
channel link(R1,R2);
  by R1,R2: request(ad:V);
            new_knowledge(l:L_set; k:K_set);
            response(l:L_set; k:K_set; n:integer) ;
 module node activity (ngh: neighbour_representation; nad:V) ;
 ip s_neighbour: array[V] of link(R1);
   r_neighbour: array[V] of link(R2);
end; {node header}
```

- local node variables *own_name, known_names,* and *knowledge* as described above,

- local node control states *idle, wait_new_knowledge, wait_response* and *termination.*

Estelle counterparts:

```
state
  idle, wait_response, wait_new_knowledge, termination;
var
  known_names : L_set;
  knowledge : K_set;
  own_name: integer;
```

- *termination_condition* that evaluates to true, in a process, if its local variable *knowledge* contains an element named by the number that equals the size of the enumerated graph.

Estelle counterpart:

```
exist i:1..max suchthat (knowledge[i].ng=graph_size)
```

- *renaming_condition* that evaluates to true, in a process, either when the node is nameless (*own_name=0*) or *knowledge* contains an element named the same as the node (i.e., the element's name equals *own_name*) but has a *stronger* (see Section 4) set of names known to it,

- *rename* procedure which assigns a new name to the node represented by the process executing the procedure; the new name is greater by 1 than all names known to neighbours; the procedure also prepares a new, unified *knowledge* for all neighbours and new *nown_names* for each of them; the new values are then communicated to neighbours.

In Estelle, both the *renaming_condition* and the *rename* procedure occur within transitions' bodies and use simple auxiliary functions. We omit them here, since their implementation has nothing specific (they are simple Pascal procedures).

A process initially has its mailbox empty, its local variables initialized as described above, and its local control state *idle.* The meaning of the guarded commands of each process is characterized as follows:

- (sending requests)
  The process broadcasts *request* to all its neighbours (asking for actual values of their local variables) and then waits for their responses changing its local

state to *wait_response*, provided: its mailbox is empty, the *termination_condition* is *false* and its control state is *idle*.

Estelle counterpart:

```
from idle to wait_response
    name sending_requests:
    begin
     all i: V do
      if ngh[i]=1 then output s_neighbour[i].request(myad);
    end;
```

- (responding to requests)

The process receiving the *request* message from a neighbour (the message is in its mailbox) responses to the requesting process sending the current values of its local variables (message *response*) provided its control state is *termination* or it is *idle* and the *termination_condition* is *false*; in the first case the control state remains the same, while in the second it changes to *wait_new_knowledge*; in any case the *request* message is removed from the mailbox.

Estelle counterpart:

```
from idle to wait_new_knowledge
 any i: V do
  priority low
    when r_neighbour[i].request(ad)
     name response_to_request_i:
     begin
      output s_neighbour[ad].response(known_names, knowledge,
              own_name);  requesting :=ad
     end;
```

{ Transition priorities plus common queue organization give a possibility of excluding the non-determinism between transitions. It is explicity excluded in the model description by guards that are impossible to express in Estelle, e.g., emptiness of a queue is not expressible. It is also worth to note that, in Estelle, a message received and served by a transition is, by definition, removed from the appropriate queue. This is not the case in the original asynchronous model operating on a mailbox, hence it must be repeated when needed}

- (receiving responses, creating and distributing knowledge)

If *response* messages arrived to the process mailbox from all neighbours while the process's control state is *wait_response*, then the process checks the *renaming_condition* and if it is satisfied, it performs the *rename* procedure; independently of the condition value the process unifies its current *knowledge* with that included in *response* messages and then this new *knowledge*, with

possibly new *known_names*, for each neighbour (message *new_knowledge*), are transmitted back to neighbour processes; finally the process removes all *response* messages from the mailbox and changes its control state to *idle*.

Estelle counterpart:

```
from wait_response to idle
   provided count=ngh_number  {i.e., all responces already arrived }
    name creating_and_distributing_knowledge:
      begin
       count := 0;
       if (own_name=0) or (exist j:1..max suchthat
           rename_cond(knowledge[j], known_names, own_name))
        then
          begin
           maxname(maxn); new_know(maxn); own_name:=maxn+1
          end;
       all i:V do if ngh[i]=1 then add(resp_k[i], knowledge);
       all i:V do if ngh[i]=1 then {i.e., if i is a neighbour}
         output s_neighbour[i].new_knowledge(resp_l[i], knowledge);
      end;
```

• (receiving new knowledge)

The process receiving *new_knowledge* message from a neighbour (the message is in its mailbox) either ignores it, if it was already in the *termination* state (and stays in the same state) or, being in the control state *wait_new_knowledge*, it actualizes its local variables with the received values and changes its control state to *idle*; then, as always, it removes the *new_knowledge* message from the mailbox.

Estelle counterpart:

```
      from termination to termination
         any i: V do
          when r_neighbour[i].new_knowledge
           name ignoring:
           begin end;
   from wait_new_knowledge to idle
        when r_neighbour[requesting].new_knowledge(l, k)
         name absorbing_new_knowledge:
         begin
          known_names := l; knowledge := k; requesting := 0
         end;
```

• (terminating)

The process passes to the control state *termination* as soon as its control state is *idle* and the *termination_condition* is satisfied.

```
Estelle counterpart:
        from idle to termination
         provided (exist i:1..max suchthat (knowledge[i].ng=graph_size))
          priority highest
           name terminal:
           begin end;
```

It seems completely obvious that connectivity (of enumerations determined by values of *own_names* variables) is preserved by anyone of the above transitions. Hence, (variants of) properties 1-5 could be proved by similar techniques as in (Mazurkiewicz, 1997) (of course, with all changes resulting from the change of model and differences between the chosen asynchronous model and this of Estelle). However, it is worth to point out a slight difference in properties which does exist. To prove the termination property of EP1 a global fairness in computation is needed. Without it, infinite and useless sequences of requests and responses are possible even if graph is not ambiguous. In the Mazurkiewicz protocol checking the **if** conditions do not require any actions, hence no useless actions are initiated. In case of an ambiguous graph such endless computations repeating exchange of information without further progress in enumeration are reflecting the fact that locally one cannot recognize the ambiguity. The global termination is locally known to all processes since a process that achieves its *termination* control state knows that eventually all other will end up in this state.

3 ALWAYS TERMINATING AND ECONOMIC PROTOCOL

The distributed enumeration protocol that is described in the previous section (and implemented in Estelle) has some flows that are inherited from its prototype in (Mazurkiewicz, 1997).

First of all, the amount of information that is exchanged between processes. For instance, the *knowledge* value is always included in a *response* message. The inclusion occurs because the description tries to follow, in an exact way, the original solution. The question is, however, whether it is necessary. If the answer was negative, then all protocol messages can be bounded by the graph size and not by the square of this value as it is in EP1.

Second, one would like a distributed protocol that always terminates, either successfully or not, as it is the case in the original. Moreover, this termination condition should not be dependent upon a fairness property which in its nature is of a global character. Recall, that this fairness property is hidden in the original solution by the assumption that makes value of the enabling condition *knowledge_in_the_neighbourhood_is_not_the_same* instantaneously "known" to the processes without any explicit evaluation procedure.

The protocol specified below (EP2) satisfies the above postulates and preserves all properties of the original solution.

Enumeration Protocol (solution) 2 (EP2)

All initial assumptions of EP1 remain the same for EP2 with the following exceptions and/or additions:

- two messages are added to the set of possible exchange messages (and the contents of the remaining messages are not exactly the same as before):

done - the message, spread over the network, causes a global and successful termination of the protocol,
trans_knowledge - the message transmits a name and a set of names (as also do now *new_knowledge* and *response* messages) and serves to ca newly created name of a process-node, together with names known to it,

- *termination_condition* reduces now to the equality: *own_name=graph_size,*

- *rename* procedure is reduced to produce a new name of a process.

Of course, there are many other details that are different in both solutions, i.e., EP1 and EP2, at the level of Estelle specification. Below we present the specification of types and messages of EP2 to enable the reader to grasp these minor differences. Anyway, the essential difference is in the size of messages (values of all neighbour *knowledge* variables are not transmitted) and the way new knowledge is disseminated. These differences in both solutions can be captured examining respective guarded commands of each node-processes. That is why we present guarded commands of EP2 in the style used for EP1, but Estelle transitions of EP2 are not presented here.

```
L_set = set of 0..graph_size;
K_el = record
         ng : integer;
         known: L_set;
       end;
K_set = ^KK_el;
KK_el = record
          kel: K_el;
          next: K_set
        end;
channel link(R1,R2);
by R1,R2: done;
          request(ad:V);
          new_knowledge(el: K_el);
          response(el: K_el);
          trans_knowledge(ad:integer; el: K_el);
```

Guarded commands for EP2: (we try to keep similar wording as in EP1 to facilitate a comparison):

- (sending requests)
  The process broadcasts *request* to all its neighbours (asking for actual values of their local variables) and then waits for their responses changing its local state to *wait_response*, provided: its mailbox is empty, the *termination_condition* is *false*, the *renaming_condition* is *true*, and its control state is *idle*.

- (responding to requests)
  The process receiving *request* from a neighbour (the *request* message is in its mailbox) responses to the requesting process sending the current values of its local variables (message *response* which now includes the values of *own_name* and *known_names* only) provided its control state is *idle* and the *termination_condition* is *false*; the new control state is *wait_new_knowledge* and the *request* message is removed from the mailbox.

- (receiving responses, creating and distributing knowledge)
  If *response* messages arrived to the process mailbox from all neighbours while it is in *wait_response* control state, then the process performs the *rename* procedure and its new *own_name* together with its current value of *known_names* is broadcast to neighbour processes (*new_knowledge* message); finally the process removes all *response* messages from the mailbox and changes its control state to *idle*.

- (receiving new knowledge)
  The process, being in the control state *wait_new_knowledge* and receiving *new_knowledge* message (the message is in its mailbox) from a neighbour previously requesting the values of the process's local variables, actualizes its local variables *knowledge* and *known_names* with the received values, changes to *idle*, removes the *new_knowledge* message from the mailbox and broadcasts two *trans_knowledge* messages: one with the values as they were received in the *new_knowledge* and the second one that includes its own values *own_name* and actualised (according to the *new_knowledge*) *known_names* (of course, the first *trans_knowledge* message is not returned to the process sending the *new_knowledge* information).

- (transit of knowledge)
  The process, being in any control state and receiving a *trans_knowledge* message, checks whether the transiting knowledge is or is not known to it (the pair of values is or is not included in its *knowledge*) and if it is not, then it actualizes its own variable *knowledge* and retransmits the *trans_knowledge* message to all its neighbours but the sending one; otherwise, i.e., when it

already knows the transiting knowledge, it ignores the received information; in both cases the message is removed from the mailbox.

- (terminating)
  The process passes to the control state *termination* if only it was in the control state *idle* and either
  the *termination_condition* was satisfied or the message *done* is in its mailbox; in both cases the process sends then the message *done* to all its neighbours.

Essential change in EP2 with respect to EP1 is in that *trans_knowledge* messages are introduced and treated. In EP1, new knowledge was acquired by repeated requests to neighbours whether it was necessary or not. In EP2, new knowledge is spread over the network only if it is really new. An information included in a *trans_message* that is already known to a process is not retransmitted. That way a new information eventually reaches all processes but also its retransmission eventually ends up, i.e., there are no useless and endless loops caused by retransmissions. A *trans_message* is only created if the enumeration is successful. In that situation no other process but the terminating one may proceed. All other processes are in their idling states waiting for a message that can trigger an action. If *done* message arrives, it is retransmitted only once and the process terminates definitely. Therefore, eventually all processes terminate. Anyway, EP2 always terminates, as did the original protocol in (Mazurkiewicz, 1997), either successfully enumerating the graph (all processes are in their *termination* control state) or without a successful enumeration, with all processes waiting forever in their *idle* state.

Again, it seems obvious, that connectivity invariant holds for EP2. The proof of it, provided we've already proved EP1, is by checking whether the invariant is preserved by new transitions. The point is that those changes we are introducing result in better protocol but do not destroy the essence of our correctness proof.

4 IMPOSSIBILITY RESULTS AND ALWAYS SUCCESSFUL PROTOCOL

The reason why a distributed enumeration protocol can fail, i.e., why it can end up without a proper enumeration of a graph, is due to some graph-theoretic property of the graph itself. This property establishes a class of graphs (networks) for which so called impossibility results hold (see, e.g., (Litovsky at al., 1993)). This class of *ambiguous* graphs is defined by a notion of *graph folding* which is a graph morphism such that its restriction to any graph node and its neighbourhood is bijective. Recall, that a graph morphism is a mapping from the set of nodes of the graph into the set of nodes of another graph preserving graph relation (graph edges). Now, a graph is ambiguous if there exists a folding (from the graph nodes onto the nodes of another graph) that is not an isomorphism.

A nice characterization of the class through the notion of graph enumeration (labelling) is given and exploited in (Mazurkiewicz, 1997). Let $G = (V, R)$ be a graph. Denote by $N(x)$ the set of all neighbours of the node x in V. Recall, that any function $f : V \to \{1, 2, \ldots, |V|\}$ is called *enumeration* of the graph G (the enumeration f is *proper* or *successful*, if $f(V) = \{1, 2, \ldots, |V|\}$). Enumeration f is called *locally unique*, if for each $x, y, z \in V$ such that $f(y) = f(z)$, we have:

a. if $y, z \in N(x)$, then $y = z$
b. $f(N(y)) = f(N(z))$.

Now, a proposition from (Mazurkiewicz, 1997) says that graph $G = (V, R)$ is ambiguous if and only if there exists a locally unique and not proper enumeration of G.

In Fig. 4.1. an example of an ambiguous 6-node graph is given. A folding into a 3-node graph is depicted (dotted lines) together with a locally unique node enumeration of the bigger graph that corresponds to the folding. Both prove the ambiguity of the graph.

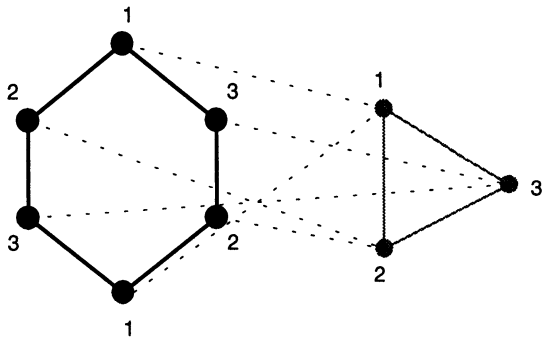

Fig. 4.1.

It is an easy exercise to get this (unsuccessful) enumeration using commands of the algorithm EP1 or EP2 in such a way that the enumeration proceeds as follows: 1, 1, 2, 2, 3, 3. It is then clear that once the above enumeration was achieved there is no hope for a successful termination. In case of EP1, only requests are possible (sending request command) but the responses are always the same, while EP2 terminates (deadlocks) in a state in which no process can continue.

The impossibility results explain that a process (node) cannot locally distinguish itself from another one that looks exactly the same in terms of attributes describing the process (node) itself and its knowledge about all neighbours. In terms of the protocol constructs, this impossibility is reflected in the formulation of the *renaming_condition.* If the attributes are equal, the

renaming_condition remains *false* and no renaming can occur. This is exactly what happens, for the example in Fig.4.1. in the situation when the given enumeration was achieved. Existence of a linear order between information characterizing processes (nodes) of the same name is absolutely necessary to start renaming and to prevent the situation when all of them start their renaming procedure in parallel. At least one of the processes should maintain the current name to hope for a successful enumeration. In (Mazurkiewicz, 1997) this order, used in the *renaming_condition,* is of lexicographical type between the set of names known to nodes.

In view of the impossibility results, it seems hopeless to think of an always successful protocol. It is true in absolute terms. However, any computation has it own dynamics (ordered history) independent of the problem that is being solved. It may be used to add order to elements that otherwise are identical.

Imagine that every new name created for a process (node), is "stamped" with a value from an ordered set of values (e.g., integers) and that the probability of the same "stamps" to the same name is arbitrary small. Nodes that are equal in previous sense become now ordered by their "stamps". A good random number generator in each node gives a sufficient practical solution for generating of "stamps". This is the whole idea of an enumeration protocol that terminates successfully with arbitrary high probability. Hence "always " terminating in probabilistic terminology. By a slight modification of EP2 we get always successfully terminating enumeration protocol EP3. Its Estelle specification consists in a trivial adaptation of internal, primitive procedures of EP2.

Repeating the exercise performed previously with EP2 for the example in Fig.4.1, will show that with the enumeration given in this figure and achieved by the protocol EP3 there is a continuation since we have assumed, in the protocol, that stamps in such situations are always different. Therefore, there is always a node (process) for which the *renaming_condition* is satisfied. Otherwise, EP3 behaves exactly as EP2, hence all properties satisfied for EP2 are valid for EP3 as well.

5 CONCLUSIONS

We have studied the enumeration protocol given in (Mazurkiewicz, 1997) and provided a version (EP1) of it that does not use shared variables but only exchanges information by messages. This protocol is then implemented (specified) in the protocol specification language Estelle. Atomicity of actions in the new protocol is of smaller "granularity". An improved version (EP2) was then proposed which is economic in message exchange and does not assume a computation fairness implicitly present in the original solution and explicitly in EP1. This new version has exactly the same properties as the original one and solves also such problems as leader election and graph recognition.

A discussion on the source of non-existence of distributed enumeration algorithm for ambiguous graphs, shows its "static" character, i.e., it refers,

roughly speaking, to impossibility of distinguishing, by local means, two graphs of different size but locally isomorphic (one graph can be "fold" into another). In any enumeration algorithm, this property translates into a possibility of reaching, from the initial state, a state in which all nodes (processes) are enumerated, but there are nodes that together with their neighbourhood are isomorphic not only with respect to the graph relation but also with respect to achieved enumeration. Such nodes cannot be distinguished, by local means, hence the enumeration procedure deadlocks with the graph not properly enumerated. However, the execution history can make the two nodes distinct even in this statically ambiguous situation. It is enough to "stamp" every newly produced name by a randomly drawn number. This simple idea has been implemented to obtain, in Estelle, a distributed enumeration algorithm that always successfully terminates, if the word *always* is understood in a probabilistic sense, i.e., the success can be assured with arbitrary high probability. The algorithm, EP3, is a simple extention of EP2 described earlier.

The paper describes an exercise (case study) of a protocol design in Estelle starting from very high level, mathematical specification and ending up with its improved and executable prototype. It shows a starting, mathematical solution makes correctness proof much easier though the solution is neither efficient nor immediately executable. Then, in small steps one can obtain an improved version of the protocol that can be almost directly mapped into such a specification language as Estelle. The gain in such design process is that the final result is correct by virtue of the original proof for a mathematical formulation and correctness preserving development steps. Protocols EP1, EP2 and EP3 are all results of such steps.

REFERENCES

Bougé L. (1988), On the Existance of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes, *Acta Informatica*, **25**, 179-201.

Budkowski S. (1992), Estelle Development Toolset, *J. of Computer Networks and ISDN Systems*, **25.**

Dembiński P. (1996), Distributed and Always Successful Enumeration Algorithm, IPI PAN Reports **812.**

Dembiński P. (1998), Randomized Enumeration, IPI PAN Reports **848.**

ISO 9074 (1989), Information processing systems - Open Systems Interconnection - Estelle - A formal description technique based on an extended state transition model.

Litovsky I., Métivier Y., Zielonka W. (1993), The power and limitations of local computations on graphs and networks, *Lecture Notes in Com. Sci.*, **657**, 333-345.

Mazurkiewicz A. (1997), Distributed Enumeration, *Information Processing Letters* **61**, 233-239.