Piotr Dembiñski

A distributed algorithm for dynamic resource allocation: development and validation in Estelle*

Nr 907

Warszawa, marzec 2000

Abstract

This paper describes the development and validation of an Estelle specification of a distributed algorithm solving the dynamic resource allocation problem. This is a modified version of the algorithm originally proposed in [Winkowski 1998]. The specification has been obtained and validated with Estelle Development Tools. The main steps in the construction of the specification are presented and the simulation techniques validating the obtained specification are sketched. The simulations show that the specification is free of dynamic errors and preserves such properties as mutual exclusion between tasks requiring the overlapping sets of resources and starvation freedom - the properties essential for any correct solution of the resource allocation problem. The full Estelle version of the algorithm is given in the Appendix.

Key words: distributed systems, multi-agent systems, distributed algorithms, resource allocation problem, Estelle specifications, validation, simulation.

Streszczenie

Rozproszony algorytm dynamicznej alokacji zasobów: specyfikacja i weryfikacja w jêzyku Estelle

W pracy opisano przebieg specyfikowania w jêzyku Estelle, a nastêpnie symulacyjnej weryfikacji zmodyfikowanej wersji rozproszonego algorytmu (zaproponowanego w pracy [Winkowski 1998]) rozwi¹zuj¹cego problem dynamicznej alokacji zasobów. Zarówno konstrukcjê specyfikacji, jak i jej weryfikacjê przeprowadzono z pomoc¹ narzêdzi Estelle Development Toolset. Pokazano g³ówne kroki w tworzeniu specyfikacji oraz naszkicowano techniki symulacjne u¿yte do jej weryfikacj. Symulacja pozwoli³a na stwierdzenie braku dynamicznych b³edów specyfikacji, ale przede wszystkim pokaza³a takie podstawowe w³asnoœci dla zapewnienia poprawnoœci ka¿dego rozwi¹zania problemu dynamicznej alokacji, jak wykluczanie siê zadañ ubiegaj¹cych siê o te same zasoby i brak tzw. "zag³adzania" zadañ. Pe³ny tekst specyfikacji algorytmu w jêzyku Estelle zosta³ zamieszczony a Dodatku.

S³owa kluczowe: systemy rozproszone, systemy wieloagentowe, algorytmy rozproszone, problem alokacji zasobów, specyfikacje Estelle, weryfikacja, symulacja.

---

* Partially supported by the Esprit Project 20288 CRIT-2 and INCO-Copernicus Project 977124 IDECOMP

Pracê zg³osi³: Antoni Mazurkiewicz

Adres autora:
Piotr Dembiñski
Instytut Podstaw Informatyki PAN
ul. Ordona 21
01-237 WARSZAWA
e-mail: piotrd@ipipan.waw.pl

Symbole klasyfikacji rzeczowej: C.2.4, D.4.1

Na prawach rêkopisu
Printed as manuscript

Nak³ad 140 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w marcu 2000 r.
ISSN 0138-06-48

1. Introduction

This paper describes the development and validation of an Estelle ([ISO 9074]) specification of a modified version of the distributed resource allocation algorithm proposed in [Winkowski, 1998]. The specification has been obtained and validated with Estelle Development Tools (EDT, [Budkowski, 1992]).

The algorithm solves the dynamic resource allocation problem. It means that it deals with a generalization of: the original Dijkstra dining philosophers problem [Dijkstra, 1971], the drinking philosophers problem as defined in [Chandy and Misra, 1984], and many variants of the dynamic resource allocation problems as considered e.g., in [Awerbuch and Saks, 1990], [Choy and Singh, 1993], or [Rhee, 1998]. All these problems deal with processes that share recources and run concurrently. This can lead a conflicting situation in which some of them request simultaneously overlapping sets of resources. An allocation algorithm solves these conflicts, without limiting the possible concurrency and without any global scheduler, in such a way that each of the conflicting processes eventually gets the required resources. In Dijkstra formulation both the set of processes and the set of resources requested by a process are fixed. The drinking philosophers problem allows a process to request different sets of resources, but the maximal set of potentially conflicting processes is known a priori. In the dynamic allocation algorithms the set of processes may change too but, usually, some a priori knowledge of the conflicting processes are known. The modular approach proposed in [Rhee, 1998] is free from the last assumption but it constructs its solution using known allocation algorithms as subroutines (the modularity is understood similarly as, e.g., in [Chandy and Misra, 1984] and others). Hence, the dynamism is restricted by the subroutine solution that requires a fixed initial configuration to start with.

The solution proposed by [Winkowski, 1998] works without any of the above constraints. The processes realizing tasks (or simply tasks) that request resources are controlled by a number of agents. The tasks assigned to agents arrive continuously. An agent requesting resources for a task identifies the the task by a numbered ticket from the set of tickets in the agent's disposition. That way, the global number of competing tasks is limited at a given moment by the number of initially distributed tickets in the system. The competing tasks are ordered by the tickets' numbers. The last assumption is important and, to some extent, replaces the requirement occurring in many other solutions that the precedence graph of the processes is acyclic (see e.g., [Chandy and Misra, 1984]). The task's ticket is released when the task received (booked) all needed resources. The ticket can be immediately assigned by the agent to a new, waiting task. While the informal description of the algorithm in [Winkowski, 1998] is presented in terms of asynchronous message passing model, its formal correctness proof deals with a global transition system. Transitions in such system implicitly assume e.g., some global knowledge to check certain non-local conditions or indivisibility of some network operations. For that reason, it is sometime doubtful whether the algorithm can work properly in a truly distributed environment without, e.g., some time synchronization between agents (a synchronization is explicitly suggested). One of the aim of this paper is to show that a modified version of the algorithm can be successfully realized when only the message passing mechanism is present. That's why, among others, Estelle has been chosen as a description language. Additionally, the powerful EDT tools (especially the Estelle simulator/debugger) offered opportunities to quickly experiment with (versions of) the algorithm.

Estelle is based on a (well known) model assuming that each process (module instance) is described by a finite set of *guarded commands* (called *transitions*). A *guard* is a boolean condition over the space of the process *internal states* and the contents of its (unbounded) *communication (FIFO) queues* storing incoming *messages*. There may be only one queue for all input messages, but there is also a possibility of separating inputs from different correspondents. A *transition* defines an *action* that may change an internal state, removes the first message from the process's queue (if the guard refers to the message), 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 input queue. The action defined by a module transition is assumed *indivisible* (or *atomic*). The communication is *asynchronous* in that a module (process) may always send a message or, in other words, operations of sending and receiving messages are not synchronized. Amodule, in each of the computation steps, performs one of the transition from among those for which their guards evaluate to *true*.

In the next section (Section 2) an Estelle model of the distributed system under consideration is presented together with Estelle representation of some elements essential to formulate and understand the proposed solution of the resource allocation problem. In Section 3 the algorithm in Estelle is described in some details and in Section 4 the important steps of validation are briefly characterized together with a short summary of simulation experiments. Appendix contains a full Estelle version of the system.

2. General model, Estelle representation and assumptions

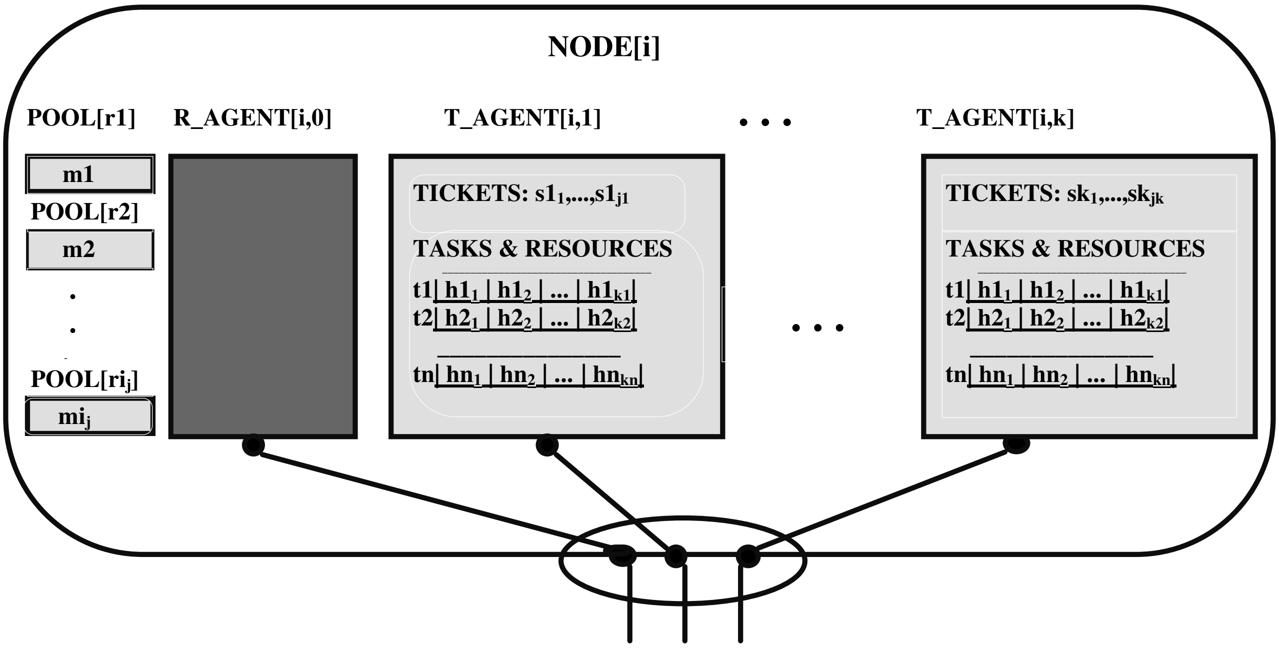
In the formulation of the allocation problem, implemented then in Estelle, it is assumed that there is a number of nodes in which different types of resources may occur. There may be a number of resources of the same type in a node. They are grouped into *Pools.* Resources (Pools) in a node (if any) are managed by the *Resource Managing Agent* (*R_Agent*) local to each node. A node has a number of (identical) *Task Managing Agents* (*T_Agents*) responsible for booking of resources needed by incoming tasks. A task that may occur spontaneously in a node is assigned to a T_Agent of this node, i.e., it is appended to the task queue of this agent. It may need any resource from those existing in the network nodes. The booking is performed by a dialogue between the T_Agent managing the task and the R_Agents managing the needed resources. The dialogue defines the core of the allocation algorithm and is described in detail in the next section. The dialogue begins when a waiting task receives a *numbered ticket* from its managing T_Agent. The tickets are distributed among T_Agents when the whole system is initialized. Agents communicate through the *network*, i.e., an agent sends an addressed message to the network which is responsible for the reliable transmission of messages between nodes (agents). It is assumed that the transmission is fair, i.e., no node is discriminated and every message sent by an agent eventually reaches its destination. The general node structure is given and explained in Fig.1 and an example of a concrete network configuration is depicted in Fig. 2.

Global assumptions about the specified system in Estelle are given in form of specific constants. In this case we have the following self explaining constants (only the first two values are known to all T_Agents):

```
const
  Net_Size = any integer;           {number of nodes in the system; in Fig.2, Net_Size = 3}
  Number_of_Resource_Types = any integer;               {in Fig.2 this number equals 3}
  Max_Number_of_Pool_Resources = any integer;           {in Fig.2 this number equals 1}
  Max_Number_of_Task_Agents = any integer;
                    {maximal number of task agents in a node; maximal number of all agents
                     in a node equals Max_Number_of_Task_Agents + 1}
  Number_of_Agent_Tickets= any integer;                  {in Fig.2 this number equals 2}
```

There are several global types specified in Estelle referring to the above constants, most of them having an auxiliary character. We list them all below. Among them the type 'Net_Pools' (in bold) is of importance because the initial value of that type, describing the quantity and distribution of available resources, is known to all agents of the specified system and the value is not changed by the allocation algorithm.

```
type
  Zero_One = 0..1;
  N_S = 1..Net_Size;
  N_A = 0.. Max_Number_of_Task_Agents;
  N_TA = 1.. Max_Number_of_Task_Agents;
  P_R = 0.. Max_Number_of_Pool_Resources;
  R_T = 1.. Number_of_Resource_Types;
  A_T = 1.. Number_of_Agent_Tickets ;
  NNet_Agents = array[N_S] of N_A ;
             { for a variable agent of type NN_ Agents, agent[i] denotes the number
               of task agents in the node i }
  Agent_Tickets = array[A_T] of integer;
  Node_Tickets = array[N_TA] of Agent_Tickets;
  Net_Tickets = array[N_S] of Node_Tickets;
             { for a variable a_tickets of type Agent_Tickets, a_tickets[i] is a
               number identifying i-th ticket assigned to an agent in a node}
  T_Plan = array[R_T] of integer;
  Node_Pools = array[R_T] of P_R;
  Net_Pools = array[N_S] of Node_Pools;
             {for a variable pool of type Net_Pools, pool[i, j] = pool[i][j]=k > 0 means that
              there is a pool of k resources of the type j in the node i;
              pool[i, j] = 0 means that there are no resources of the type j in the node i }
```



T_AGENT[i,j] j-th Task Managing Agent of the i-th node;

R_AGENT[i,0] Resource Managing Task of the i-th node

POOL[ri]

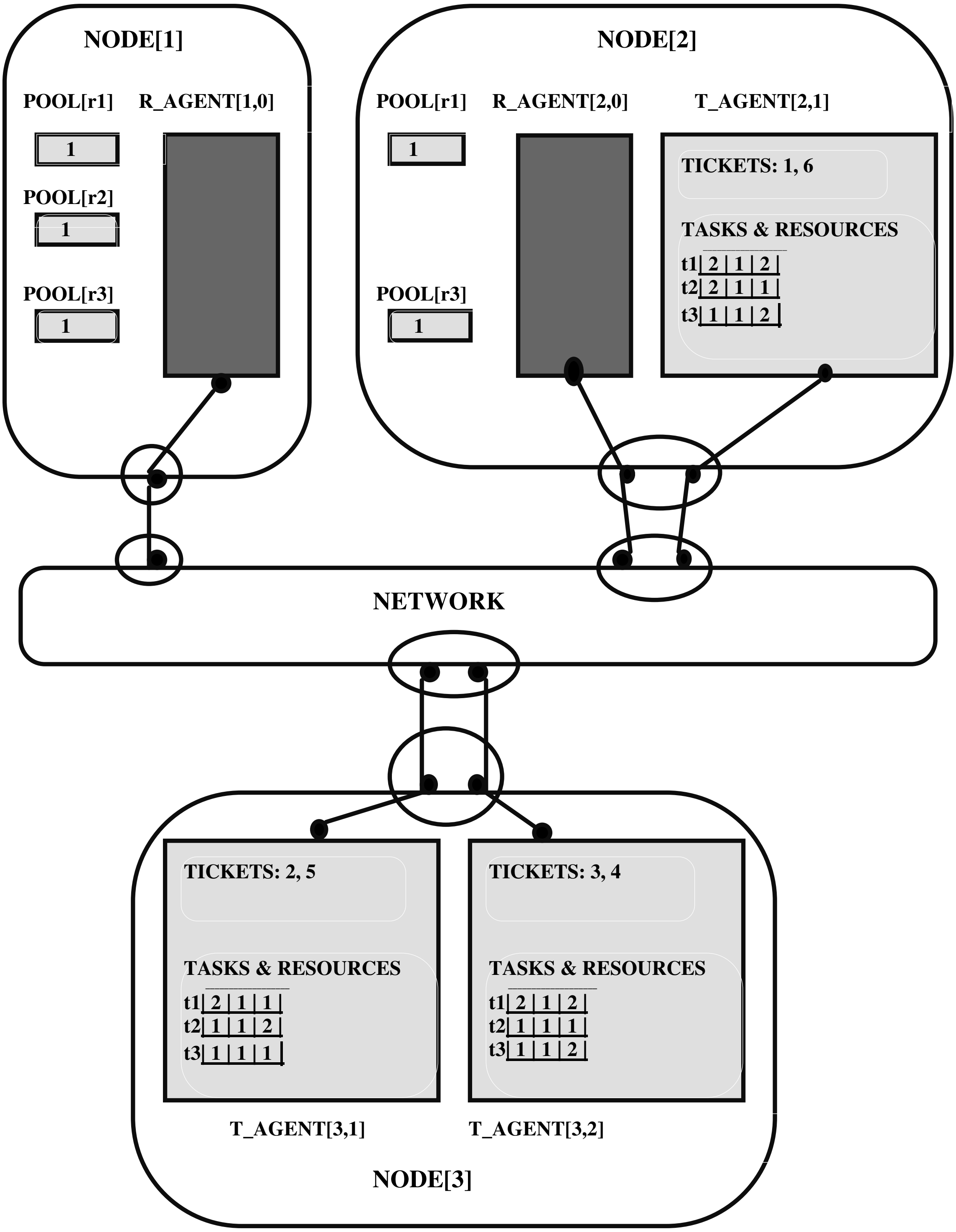
| m | Pool of size m of the resources of type ri

TICKETS: $s1_1,...,s1_{j1}$ T_Agent's tickets numbered s1, s2, ...

TASKS & RESOURCES T_Agent's waiting tasks

tj | hj_1 | hj_2 | ... | hj_{kj}| task tj requires k_j different resources (of types rj_1, rj_2, .) located in nodes hj_1, hj_2, ...

Fig.1.



t1| 2 | 1 | 1 | means that t1 requires the resource r1 from POOL[r1] in NODE[2] and resources r2 and r3 from POOL[r2] and POOL[r3] in NODE[1], respectively; each of the pools contains just one resource of the given type; NODE[3] has no resources, hence R_TASK[3,0] is not present too; each of T_agents has two numbered tickets in its disposition.

Fig.2.

Interior configuration of one node in the system (see Fig.1.) is determined by its actual parameters while the node and all its agents are initialized. The external visibility of nodes, agents and network is defined through the messages they exchange and the latter are listed within the following channel construct (the role of each message will be explained in the next section):

```
channel Link(A, N);
  by A, N:
    request(sn_ad: N_S; sa_ad: N_TA; rn_ad: N_S; r_type: R_T; t_ticket, n_req: integer);
    req_rejected(rn_ad: N_S; ra_ad: N_TA; r_type: R_T; t_ticket, n_req: integer);
    req_accepted(rn_ad: N_S; ra_ad: N_TA; r_type: R_T; t_ticket, n_req: integer);
    book(rn_ad:N_S; r_type: R_T; t_ticket:integer);
    book_confirmed(rn_ad: N_S; ra_ad:N_TA; r_type: R_T; t_ticket: integer);
    withdraw(rn_ad: N_S; r_type: R_T; t_ticket: integer);
    rrelease(rn_ad: N_S; r_type: R_T);
    broadcast_add(pl: T_Plan);
    broadcast_sub(pl: T_Plan);
    add(pl: T_Plan);
    sub(pl: T_Plan);
```

A node is specified and created (see below) as a module with the parameters specifying some global information: about the resources (‘ne_pools’), the number of the node’s T_Agents (‘n_agents’), the node address (‘n_ad’), and tickets (‘n_tickets’). Actual values of this parameters are then passed to each of the T_Agents created (and identified by a module variable specified in the ‘modvar’ section) together with the agent individual address (‘a_ad’). The R_Agent of the node (if any) gets only the information concerning the node’s resource pools (‘n_pools’). The node’s communication ports (‘node_ips’) are attached to the respective agents’ ports (‘agent_ip’) which means that communication is treated as if each agent was directly connected to the network (e.g., there is no multiplexing defined at the node level). That way agents may work really independently and concurrently.

```
module Node (ne_pools: Net_Pools; n_agents, n_ad: integer; n_tickets: Node_Tickets) ;
  ip node_ips: array[N_A] of Link(A);
end; {Node header}
body Node_Body for Node;

module Task_Agent systemactivity (n_ad: N_S; a_ad: N_TA; ne_pools: Net_Pools;
                                  tickets: Agent_Tickets);
  ip  agent_ip: Link(A);
end;
body Task_Agent_Body for Task_Agent;

module Resource_Agent systemactivity (n_pools: Node_Pools);
 ip  agent_ip: Link(A);
end;
body Resource_Agent_Body for Resource_Agent;

modvar
  t_agent_var: array[N_TA] of Task_Agent;
  r_agent_var: Resource_Agent;

initialize
 var m : integer;
 begin
   if n_agents<>0 then
     for m:=1 to n_agents do
       begin
         init t_agent_var[m] with Task_Agent_Body(n_ad, m, ne_pools, n_tickets[m]);
         attach node_ips[m] to t_agent_var[m].agent_ip
       end;
   if (exist j: R_T suchthat (ne_pools[n_ad, j]>0)) then
```

```
      begin
        init r_agent_var with Resource_Agent_Body(ne_pools[n_ad]);
        attach node_ips[0] to r_agent_var.agent_ip
      end
  end;
```

The overall structure of a given distributed system is composed of a number of the above nodes and the network specified by the module:

```
module Network systemactivity;
  ip net_ips: array[N_S, N_A] of Link(N);
end; {Network header}
body Network_Body for Network;
```

The network behavior is reduced to agent-to-agent message transmission. The following is then a typical network transition scheme (exemplified by the transition of a task managing agent sending a request for a resource of certain type ('r_type' parameter) to the resource managing agent in a node that has a pool of resources of that type):

```
trans
   any i:N_S; j : N_TA do
     when net_ips[i,j].request(sn_ad, sa_ad, rn_ad, r_type, t_ticket, n_req)
       begin
         output net_ips[rn_ad, 0].request(sn_ad, sa_ad, rn_ad, r_type, t_ticket, n_req)
       end;
```

Slightly different are the ‘broadcast‘ network transitions, i.e., those responding to the 'broadcast_add' or 'broadcast_sub' agents' messages. The messages mean that their contents is to be transmitted to all nodes (agents) in the network. Below is the 'brodcast_add' scheme:

```
when net_ips[i,j].broadcast_add(pl)
       begin
         all k: N_S; l: N_TA do
          if ((k<>i) or (l<>j)) then output net_ips[k,l].add(pl)
       end;
```

Creation of the nodes and network is specified by the global initialization. Below we give this initialization that creates the system specified in Fig.2.

```
modvar
  nodes: array[N_S] of Node;
  medium : Network;

initialize
  var i,j:integer;
  begin
{ all fields of arrays not specified below are set to 0}
{ Pools_ditribution (ne_pools); }
   ne_pools[1,1]:=1;
   ne_pools[1,2]:=1;
   ne_pools[1,3]:=1;
   ne_pools[2,1]:=1;
   ne_pools[2,3]:=1;

 { Agents_distribution (ne_agents);}
   ne_agents[2]:=1; ne_agents[3]:=2;

 { TicketDistribution (ne_tickets);}
   ne_tickets[2,1,1]:=1;
   ne_tickets[2,1,2]:=6;
```

```
 ne_tickets[3,1,1]:=2;
 ne_tickets[3,1,2]:=5;
 ne_tickets[3,2,1]:=3;
 ne_tickets[3,2,2]:=4;

 init medium with Network_Body;
 for i:=1 to Net_Size do
   begin
      init nodes[i] with Node_Body( ne_pools, ne_agents[i], i, u, r, ne_tickets[i]);
      for j:=0 to ne_agents[i] do
      connect nodes[i].node_ips[j] to medium.net_ips[i,j]
   end
end;
```

For the proper understanding of the allocation problem and its solution in the context of the distributed system described above, it is important to point out once more what is globally known in the system (known to all agents in the system) and what are initial assumptions (appropriately represented in Estelle or directly satisfied by the Estelle model).

Global, initial knowledge:

- net size, i.e., the number of nodes in the system (the constant ‘N_S’), and
- resource distribution in the system, i.e., types and quantities of resources in the network nodes.

The above initial parameters are not changed by the allocation algorithm, but information of currently available resources in the system (this information is dynamically changing) is an important factor for each agent decisions. For that reason, the task agents keep and adjust this information locally. Adjusting is based on the broadcast messages. It is important, however, to stress that agents' current knowledge on available resources is not the same and it is usually different from the real situation at a given moment. That way each agent behavior is truly independent and based only on its local knowledge.

General assumptions:

- the communication network is reliable and fair (every sent message eventually arrives to its destination)
- if a sender-agent sends the message m_1 that precedes the message m_2 to the same receiver-agent, then the messages arrive in the same order,
- every agent (in general, every module in Estelle) behaves “urgently”, i.e., when it can execute an action it starts to execute it without any unspecified delay (specified delays do exist in Estelle),
- actions of agent’s transitions (in general, transitions’ actions in Estelle) are atomic in that they last a finite amount of time, cannot be aborted and the intermediate values of their execution (as Pascal-like programs) cannot be observed and/or used,
- between the beginning and the end of execution of a transition action only a finite number of other operations in the whole system may be executed (finite delay property).

From the above assumptions it follows, among other things, that it is not possible to have a situation in which one agent is "infinitely" faster than another or, in other words, there is a computation in the system in which a permanently enabled transition is never executed.

3. Allocation algorithm in Estelle: T_Agent/R_Agents dialogue

3.1. T_Agent specification

In simulation we need tasks to be spontaneously created and assigned to agents. In the described Estelle version of the algorithm it is done by means of a separate transition included in the description of any T_Agent (transition T0 in the Appendix). From time to time, the transition creates a task and assigns (randomly) addresses of nodes posessing pools of resources of the required types. To obtain the test configuration as specified in Fig.2 the tasks were created explicitly when the appropriate agents were initialized. None of these two forms (random or explicit) is discussed here. It is assumed that there is a queue ‘wait’ (possibly empty) of waiting tasks in each T_Agent. The behavior of a T_Agent is described very briefly below. The description is devided into parts that correspond to the Estelle transitions named T1, T2, ..., T7 in the Appendix. The reader can confront these partial informal descriptions with their Estelle counterparts (T1 in Estelle is repeated below) to conclude (hopefully) that the Estelle coding is easy to read and understand:

Transition T1
Let a T_Agent have a free ticket and there is a task, in the agent's waiting queue ‘wait’, whose resource requirements can be satisfied in a current situation (according to the knowledge of the agent). The resource requirements are given by a list resource types together with (addresses of) nodes possessing pools of resources of these types. In such asituation, the agent assigns the ticket to the first task in the ‘wait’ queue for which the resource requirments can be satisfied and sends the ‘request’ message (with parameters specifying the sender and receiver addresses, resource type, and the ticket number) to all listed nodes (their R_Agents). It also broadcasts information about the demanded resources.

Transition T2
It waits for responses and if all responses are positive (‘req_accepted’), then it books all the needed resources (sends ‘book’ order to the responding nodes).

Transition T3
When the booking is acknowledged (all ‘book_confirmed’ messages are received), the task is executed and the used resources are released (‘rrelease’ message is sent to the appropriate nodes). Then the agent broadcasts the information that the booked resources have been released (‘broadcast_sub(pl)’ message with parameter ‘pl’ specifying the types and locations of resources that have been realeased). This information, when received, serves to update the local resource knowledge of each agent (‘SubCompete(pl)’ procedure). This, in turn, may enable a task, that is waiting for some of the booked resources, to start the requesting procedure. Finally, the ticket is returned to the agent. The task has been successfully served.

Transition T4
If, however, a response is negative (‘req_rejected’; the reasons of rejection are specified in R_Agent description), then immediately all sent requests for the task should be withdrawn (‘withdraw’ message is sent).

Transition T5
In this situation, the task still holds the ticket and the agent waits for a possibility (i.e,. it waits for a situation in which ‘ReqPermit’ holds) to repeat the requesting procedure.

Transitions T6 and T7
If an information broadcast by some other agent arrives (‘add’ or ‘sub’ message), then the local agent knowledge about the available resources is to be updated and this is done by ‘AddCompete’ and ‘SubCompete’ procedures, respectively.

Below we give some commented type definitions that help to understand better the details of the solution and its (correct) Estelle implementation.

```
type
   Task_RInf = (white, black, red, green);
   Task = array[R_T] of Task_RInf;
   {for a variable tsk of Task:
     - initial values of tsk shall be restricted to black and white,
     - tsk[j]=white indicates that the task needs a resource of type j,
     - tsk[j]=black indicates that the task doesn't need a resource of type j.
     - tsk[j]=green indicates that a resource of type j was already requested
        and the request is currently accepted (resource is available),
     - tsk[j]=red indicates that a resource of type j was already requested
        and the request is currently rejected (resource ic not available)}
  T_Plan = array[R_T] of N_S;
    {for a newly created variable tsk of Task and its associated plan of T_Plan,
     if tsk[j]=white, than plan[j]=i indicates that the node i has a pool
     of resources of type j, i.e. ne_pool[i, j]>0 ; several different plans are possible}
   Task_Info = record
                    ticket: integer;
```

```
                tsk: Task;
                tplan: T_Plan;
                narg: R_T           {the number of 'white' values initially in 'tsk'}
              end;
    Task_Control = array[A_T] of Task_Info;
    Ticket_Control = array[A_T] of Zero_One;
    Exec_Control = array[A_T] of Zero_One;
    Booking_Control = array[A_T] of integer;
    Req_Control = array[A_T] of integer;
    Task_Compet_Control = array[N_S, R_T] of integer;
```

The comments in the above specification should suffice to understand what type of information about a task is needed and kept to enable the dialogue between the T_Agent serving the task and R_Agents managing the resources requested by it. A number of control variables (of above types) help to keep track of this dialogue. We are going to explain in detail the meaning of two of these control variables in Section 4. We end this section with an Estelle specification of the transition T1 informally described above. With the assumption that the 'ReqPermit' boolean function returns 'true' if the "resource requirements can be satisfied in a current situation (according to the knowledge of the agent)", the implementation seems to be completely self-explaining.

Estelle transition T1:

```
trans {transition T1}
  when r_wait.wait(t_i)
    provided (exist i:A_T suchthat (ticket_ctrl[i]=0)) and (ReqPermit(t_i, tc_ctrl, ne_pools))
      name T1: begin
              TicketAssign(t_i);
               all i: R_T do
                 if t_i.tsk[i]=white then
                    output agent_ip.request(n_ad, a_ad, t_i.tplan[i], i, t_i.ticket, 1);
              AddCompete(t_i.tplan);
              output agent_ip.broadcast_add(t_i.tplan)
            end;
```

3.2. R_Agent specification

Every R_Agent is in a sense passive because it only reacts to the messages received from T_Agents. But the way it reacts is decisive for the proper behavior of the whole algorithm. We present below an informal description of the R_Agent behavior. As in case of T_Agent, the description is divided into parts corresponding to the Estelle transitions from the Appendix.

Transition R1

Each pool of resources (in a node) has an individual queue managed by R_Agent. If a 'request' message arrived with a type of resource as one of its parameters ('r_type'), the R_Agent decides whether the message (as a matter o fact its numbered ticket) is only appended to the corresponding pool queue or it is immediately accepted (hence, also appended to the corresponding pool queue) or rejected. The decision is made on the basis of results of the procedure 'Qtest'. In the acceptation and rejection cases an appropriate answer ('req_accepted' or 'req_rejected', respectively) is sent. The request is rejected if the number of waiting elements (requests) with ticket numbers lower than the one of the arrived message is greater or equal than the number of currently free resources in the pool. Otherwise (the number is lower), the request is either accepted (if, additionally, the length of the queue is less than the number of free resources) or only appended to the queue (the acknowledged acceptance is postponed).

Transition R2

In response to a 'withdraw' message, R_Agent removes the request (identified uniquely by its ticket number - one of the message parameter) from the proper pool queue and all requests following the removed one in the queue are advanced one position. Now, the first of the advanced requests that was not accepted yet (if any) can be accepted ('req_accepted' is sent).

Transition R3

For a ‘book’ message R_Agent responses with ‘book_confirmed’, removes the corresponding request from the pool queue, and decreases the number of free resources of the pool by one.

Transition R4

Finally, if a ‘rrelease’ arrives, the previously booked resource is released, i.e., the number of free resources in the corresponding pool is increased by one. Similarly as in case of ‘withdraw’ message, the first request in the pool queue (if any) from those not accepted yet is accepted.

Similarly as in case of T_Agent, we add below the Estelle specification of one of the transitions (R1) that is informally described above. Again, the implementation is self-explaining with the assumption that, after the ‘Qtest’ procedure has been executed, ‘qlength’ and ‘count’ values properly define, “the number of waiting elements (requests)” and “the number of waiting elements (requests) with ticket numbers lower than the ticket number of the arrived message”, respectively.

Estelle transition R1:

```
trans {transition R1}
  when agent_ip.request(sn_ad, sa_ad, rn_ad, r_type, t_ticket, n_req)
   name R1: begin
              QTest(pool_ctrl[r_type], t_ticket, count, qlength);
              if count<free_res_ctrl[r_type] then
               begin
                 qt_elem.tnad:=sn_ad; qt_elem.taad:=sa_ad; qt_elem.ticket:=t_ticket;
                 qt_elem.n_r:=n_req; qt_elem.accept:=false; qt_elem.next:=nil;
                 if qlength<free_res_ctrl[r_type] then
                   begin
                    output agent_ip.req_accepted(sn_ad, sa_ad, r_type, t_ticket, n_req);
                    qt_elem.accept:=true
                   end;
                 Append(qt_elem, pool_ctrl[r_type])
               end
               else
                output agent_ip.req_rejected(sn_ad, sa_ad, r_type, t_ticket, n_req)
              end;
```

4. Validation

A validation of the distributed allocation algorithm has to answer two correctness questions: the first asks whether the proposed solution (i.e., T_Agent and R_Agent dialogue) leads always, once a task obtained a ticket, to successful booking of the resources required by the task, while the second asks whether the Estelle implementation preserves this first correctness criterion. In the discussion below these two validation parts are not (can not be) very strictly separated. The intention is, on one hand, to explain the most important features that makes the distributed algorithm work and, on the other hand, to show how the validation exploits possibilities offered by the Estelle supporting tools, in particular the EDT interactive simulator/debugger. The correctness of the algorithm requires several partial questions to be answered. For instance, how to prevent the whole system from deadlocks that may occur when different agents (tasks) are competing for the same resources, or how to minimize the number of repeated but useless requests, for a task, in a situation when there is neither global scheduler nor any time synchronization. Recall that the only global assumption is that the communication network is reliable and fair (every message sent eventually arrives to its destination). Yet another, auxiliary question is how to distinguish responses for a current, repeated request from those still related to the previous one (for the same task when the previous request was rejected).

We begin with a quite detailed explanation of the essential mechanisms that make the algorithm (and its Estelle version) a correct solution to the dynamic allocation problem (Section 4.1). Then we sketch the role and use of the Estelle tools (EDT) in developing and validating this Estelle solution (Section 4.2).

4.1 Correctness of the algorithm

We begin with an explanation of the and role played by the most important of the control variables:

- 'tc_ctrl' of the type 'Task_Compet_Control = array[N_S, R_T] of integer'.

Then, we continue to explain the algorithm's mechanism that serves to eliminate deadlock situations. Both the use of the variable and the mechanism are essential for understanding how the algorithm (and its Estelle implementation) ensures the eventual successful booking of requested resources for every task. The control variable serves also to minimize the number of repeated but useless requests.

The variable 'tc_ctrl' is the one that stores the local knowledge of each T_Agent about the current demand of the resources in the whole system. Since both (addresses of) nodes and types of resources are represented by integers, the value 'tc_ctrl[i, j] = k' means that currently there is the demand for k resources of type j in the pool located in the node i, as represented in Fig.3. Initially the value of 'tc_ctrl[i, j]' equals, for every i in N_S and j in R_T, the initial capacity of the respective pool in the node, i.e., 'ne_pool[i, j]'. The initial capacities, i.e., the value of 'ne_pool', is known to all agents.

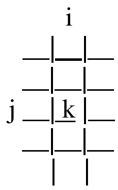


Fig.3.

The knowledge is updated when either a message 'add(pl: T_Plan)' or 'sub(pl: T_Plan)' have come to the agent, meaning that another agent had requested or released, respectively, resources (of types and locations) specified by the message's parameter 'pl'. Appropriate values 'tc_ctrl[i, j]' are increased or decreased by one. The same effect have procedures 'AddCompete(pl: T_Plan)' or 'SubCompete(pl: T_Plan)' performed by the agent when it sends, for a given task, messages requesting or releasing resources, respectively (see transitions T1 and T4). Since the broadcast messages come to different agents at different moments, their global knowledge is usually not the same and not really up-to-date. However, this knowledge is used to check (function 'ReqPermit') whether a task's "resource requirements can be satisfied in a current situation" as it was formulated earlier and, consequently, whether the (first or subsequent, repeated) 'request' message can be sent. Obviously, the resource requirements can be satisfied for a task if, for each of its required resource of type 'j' located in node 'i', 'tc_ctrl[i, j]< ne_pool[i, j]', i.e., the currently known demand for the resource in this node is less than the initial capacity of the respective pool in the node.

The above detailed description allows for understanding how the number of repeated but useless requests are minimized and possible infinite races eliminated. It also is a basis for understanding and proving that a task which obtained a ticket is eventually successfully served (we abstract here from the problem of how deadlocks are eliminated - this problem is discussed next in this section). The idea of such a proof can be presented, without loosing the generality, for the simplest case when there is only one resource for which a number of tasks compete. All of them can start their requests independently, but if all of them do it (almost) simultaneously, then some have to withdraw their requests and wait for the next opportunity. Which one of them, it is decided by conditions in the transition R1. Nevertheless, these tasks that have withdrawn request still have their tickets and they all had already sent, to the other nodes (agents), the message ('broadcast_add') informing that they are in competition. Because the network is assumed fair and reliable these messages (or rather massages 'add' caused by the former) eventually reach the other nodes. When such broadcast message arrived to a T_Agent, then theagent's local 'tc_ctrl' value must increase by one. If the value is 'n', it can only reach zero again - permitting the agent to request again the resource for a task - if 'n' messages 'sub(pl)' have arrived subsequently. This, in turn, can happen only if 'n' tasks competing for the resource had booked and released it. With these assumptions, the execution of a task that started requesting the resource cannot be blocked forever even if the execution speed in other nodes is much greater.

The simple queue mechanism is not sufficient to eliminate deadlocks that are possible if tasks compete for more than one resource. The above complicated accept/reject and book/withdraw procedures are needed additionally. To explain the reason, consider again a simple example of two tasks, t1 and t2, both requiring two different resources, r1 and r2. Assume the ticked numbered 1 has been assigned to t1 and numbered 2 to t2. For i=1,2, the T_Agent managing ti sends 'request(...r1, i,...)' and 'request(...r2, i,...)' to R_Agents, say RA1 and RA2, managing resources r1 and r2, respectively. Suppose, 'request(...r1, 1,...)' arrived first to (the input queue of) RA1 and 'request(...r2, 2,...)' to RA2. Then, the remaining two requests have been received by these agents and this situation is depicted in Fig.4.(a). According to the description of the transition R1 of each R_Agent, both 'request(...r1, 1,...)' and 'request(...r2, 2,...)' are accepted and appended to the respective pool queues (the number of waiting elements in these queues was 0 and is now 1, and the number of free resources in the pools was, and still is, 1). In this new situation, neither the 'request(...r1, 2,...)' nor 'request(...r2, 1,...)' can be accepted (one request is already waiting for one free resource in each pool queue). If none of them was rejected, then we have a deadlock situation. If both were rejected, then another deadlock situation is achieved in a few steps. Therefore, one request should be rejected and another one only appended (acceptance is postponed). From the definition of the transition R1, 'request(...r1, 2,...)' is going to be rejected, because the ticket number of the request preceding it in the queue is lower and there is only one free resource in the pool. The rejection message 'reject_req(...r1, 2,...)' will force the T_Agent managing t2 to send 'withdraw(...r2, 2,...)' to RA2. It will be served by the transition R3 of this agent and will: remove 'request(...r2, 2,...)' from the queue, advance 'request(...r2, 1,...)' one position, and send 'accept_req(...r2, 1,...)'. The situation is depicted in Fig.4.(b). Now both resources can be booked for the task t1. Immediately after the task t1 has been executed, the resources can be requested and booked for t1.

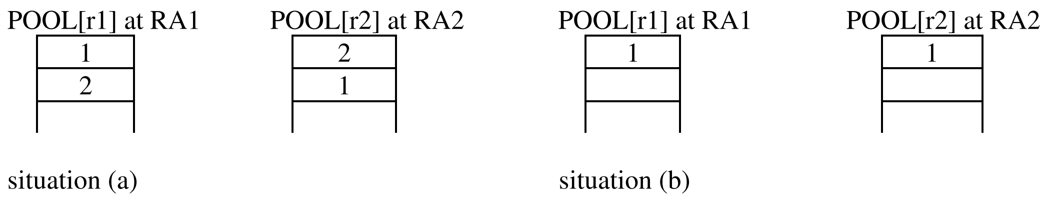


Fig.4.

As we could see the role of the 'tc_ctrl' control variable is very important in the whole algorithm. The role of other control variables have also important, but rather auxiliary character. It is illustrated, by the variable

- 'req_ctrl' of the type 'Req_Control = array[A_T] of integer'

which serves to distinguish responses for a current resource requests, for a task, from those responding to the previous request attempts for the same task. Such responses may be still arriving but should be ignored once the first rejection has been received.

The use of 'req_ctrl' is very simple. For an i-th ticket in possession of a T_Agent, 'req_ctrl[i]=k', where k>0, means that the task to which the ticket is assigned has requested resources k times. The actual value of the 'n_req' parameter is k for all 'request' messages in the last repetition. Hence, the same parameter ('n_req') in any message responding to the requests ('req_accepted' or 'req_rejected') must have the same value k. Any responding message with this value less than k is ignored.

4.2. Validation with EDT tools by simulation

Validation of any computer program begins with the coding phase when syntactical correctness is being checked. This concerns Estelle specifications as well, but we are not going to describe this phase since our interest is in verifying behavioral correctness of the Estelle specification of the allocation algorithm. Therefore, we concentrate on simulations after the Estelle text has been accepted and initialized by the EDT interactive simulator. According to Estelle semantics, the unit of execution (one execution step) is a transition. The user may control, observe and trace the execution steps of a specification by means of the simulator commands. Basically, any systematic validation by simulation consists of two phases: step-by-step, user controlled simulations and continuous, extensive simulations automatically controlled by predefined scenarios. The first phase prepares (and enables) the second. The validation is considered successfully completed if long continuous simulation sessions terminated without detecting errors. Below we sketch the two simulation phases while validating the allocation algorithm. It is not our intention to describe (or to advertise) the used Estelle tools. Therefore, we are not going to present the user-friendly graphical interface of EDT, various menus and other tools' facilities. We are concentrating on essential steps that have led to fully automatic simulations validating the algorithm behavior.

Step-by-step simulation

The simulation was based on the algorithm understanding as described in previous sections. With this in mind, in each step, the possible next steps (executable transitions) were examined to select one of them, but also to check whether all of them were justified in a given situation. That way a rough correctness of the transitions' enabling conditions was verified. Once the next step (transition) had been selected and executed, the obtained state of the system was carefully examined to verify the result of the execution. It was done for each transition, in different situations, therefore the correctness of the transition actions and used by them functions and procedures was tested. The step-by-step simulation enabled to create these different situations and to force the required sequences of executions.

Let us consider, as an example, the transition T1. Assume that it was selected and executed by a T_Agent, i.e., by a module instance in Estelle representing the agent. It seemed important to verify whether the 'ReqPermit' function and/or 'AddCompete' procedure, occurring in the transition, were properly used. One may notice also that the 'PlanCreation' procedure from the transition T0 (see Appendix) could be indirectly examined in the same time. It was enough to look at values of variables (parameters) 't_i', 'tc_ctrl' and 'npool' of the module instance under consideration to conclude the following:

(a) if, for some $1 \leq j \leq$ R_T', 't_i.tsk[j] = white and (t_i.tplan[j] = 0 or t_it.plan[j] > R_S)', then there is an error in 'PlanCreation' procedure, because 't_i.tplan[j]' should point to the address (integer between 1 and R_S) of a node that has a pool of resources of type 'j', since 't_i.tsk[j] = white' indicates that the task requires that type of resource.

(b) if 't_i.tsk[j] = white' and '$1 \leq$ t_i.tplan[j] $\leq$ R_S' but 'tc_ctrl[t_i.tplan[j], j] > npool[t_i.tplan[j], j]', then there is an error in 'ReqPermit' function or in 'AddCompete' procedure, because a request cannot (should not) be sent if, to the knowledge of the agent, the capacity of some resource pool is exhausted

(c) if, for every $1 \leq j \leq$ R_T, neither (a) nor (b) holds, then the execution of T1 can be considered correct.

In order to automatically verify certain conditions in a given situation, e.g., after a transition have been executed as above, one can prepare in advance, so called, simulation "observer(s)", i.e., a sequence of simulation commands that automatically recognize the situation, verify the conditions and may, additionally, display and/or store an appropriate information. For instance, a self-explaining observer verifying (a) - (c), was created, by the command 'set_observer', as follows (each time the transition T1 was executed the observer caused the screen display of some information):

```
set_observer \
   { if $last_transition_id = T1 then\
      cwp $last_instance; #i:=0;\
      do\
         #i:=i+1;\
         if #i > R_T then print "T1 is OK" exit fi;\
         if t_i.tsk[j] = white \
         then\
            if (t_i.tplan[j] = 0 or t_it.plan[j] > R_S) then \
               break; print "error in PlanCreation procedure"\
             else\
               if (tc_ctrl[t_i.tplan[j], j] > npool[t_i.tplan[j]) then\
                  break; print "error in 'ReqPermit' or 'AddCompete" procedures"\
```

```
                fi\
              fi\
          fi\
        od\
      fi\
    }\
```

The first two lines read as follows:

"if the last executed transition was T1, then switch ('cwp') to the context of the last executed module instance (i.e., the one representing the agent that executed this transition), set an auxiliary variable #i to 0 (the simulator's "scratch-pad" identifiers are beginning with #) and...."

Similar observer commands were created for (almost) all transitions. They were then joined into one observer of the form:

```
set_observer \
   { if $last_transition_id = T0 then ... fi;\
     if $last_transition_id = T1 then ... fi;\
                  •
                  •
                  •
     if $last_transition_id = T7 then ... fi;\
     if $last_transition_id = R1 then ... fi;\
                  •
                  •
                  •
     if $last_transition_id = R4 then ... fi;\
   }
```

Continuous simulation

Once we were sufficiently convinced, by the interactive simulation, that the algorithm mechanism work correctly (what exactly means "sufficiently" is a matter of a individual decision), we switched into simulations guided by a predefined simple scenario that we are going to describe below. The general idea consisted in observing long simulation runs, each one starting with a different initial situation in which a number of tasks were created in each one of the existing 'Task_Agent' module instants. The tasks had resource requirements assigned at random. The tasks were only created in these initial situations, i.e., a variant of T0 transition was a part of the initialization phase of each 'Task_Agent' module instance and T0 was not active latter in the simulation. A run was considered successful if it achieved a deadlock situation in which all tasks were properly served. The successful deadlock could be simply described as one in which all queues in the system are empty. The simulation limit for each run should be sufficiently long, i.e., the limit number of simulation steps for one run (set by the user) must be big enough, to achieve the successful deadlock. Again, we stopped the continuous simulation if a sufficient number of subsequent runs were successful believing, that the runs exploited and tested all mechanisms of the allocation algorithm and their Estelle implementation. Of course, runs were usually not successful in the beginning. They stopped either because the run limit was achieved earlir or, most frequently, because of one of the three types of errors occurred:

- ordinary Pascal dynamic errors like: division by zero, subrange violation in expressions, etc.,
- (unsuccessful) deadlocks,
- user defined errors recognized by observers, for example, such as described earlier, concerning T1 transition.

The first two type of errors stops the Estelle simulator automatically. The command 'break' serves to stop the simulation in case of an error of the third type. Of course, in a continuous simulation, we wanted to execute several runs, one after another, without the user intervention, regardless they were or not successful. On the other hand, if the run stopped before the run limit we wanted to know what was the stopping reason to enable correction of errors. In order to make the whole scenario simple (complicated scenarios are, like programs, susceptible to errors) we decided to store as little information as was needed to be able to reconstruct latter the erroneous situations. For that reasons, we introduced the scratch-pad variable '#break_number' which was supposed to assume integer values identifying sorts of errors causing the execution of 'break' command within the above, collective for all transitions, observer. If an execution stopped, an appropriate value was assigned to the variable. Remembering the correspondence between the values and the error identifications, made it easy to recognize the reason for which the run was broken. For instance:

0 = "T1error in 'PlanCreation' procedure",
1 = "T1 error in 'ReqPermit' or 'AddCompete' procedure", etc.

Obviously, the observer parts should have been appropriately adjusted. For example, in the T1 observer part defined above, we had to replace:

'break; print "error in 'PlanCreation' procedure"' by '#break_number:=0; break', and
'break; print "error in 'ReqPermit' or 'AddCompete' procedures"' by '#break_number:=1; break'.
.
To return latter to a recognized erroneous situation it was enough to store in a file:
- the value of the run number,
- the initial value, for this run, of the seed of the simulator's random numbers generator,
- the number of so far executed simulation steps in the run, and, in case of a "break situation",
- the value of the #break_number variable.

Storing in a file ('file_display' command) is defined by the macro-commands below. One of them was executed after the simulation had been stopped for the corresponding reason. The information was stored in separated files for different type of errors. In case of deadlocks the macro stored the contents of all network queues. The overall simulation scenario - parametrized by the number of runs, '#n_runs', and the length of each run, '#run_limit' - was defined as follows :

```
macro RUN \
      { \
        #nr:=0; \
        #initial_seed:=$random;\
        do \
           if #nr = #n_runs then print "END OF SIMULATION SESSION"; exit\
              else \
                 $firing_steps:=#run_limit; continue;\
                 if ($current_firing_step = #run_limit) then RLIMIT() fi;\
                 if $is_break then BREAKS() fi;\
                 if $is_deadlock then DLOCKS() fi;\
                 if $is_error then ERRORS() fi;\
                 restart; #initial_seed:=$random\
         fi\
      od }

macro BREAKS\
      { file_display BREAKS_FILE #break_number;\
        file_display BREAKS_FILE #round_number; \
        file_display BREAKS_FILE #initial_seed; \
        file_display BREAKS_FILE $current_firing_step }

macro RLIMIT\
      { file_display RLIMIT_FILE #round_number; \
        file_display RLIMIT_FILE #initial_seed; \
        file_display RLIMIT_FILE $current_firing_step }

macro ERRORS\
      { file_display ERRORS_FILE #round_number; \
        file_display ERRORS_FILE #initial_seed; \
        file_display ERRORS_FILE $current_firing_step }
```

The macro DLOCKS below is parametrized by the number of module instances, '#n_instances', of the Estelle specification. Each module instance is identified by the unique number from the closed interval [1, #n_instances]. The numbers are easily available by the command 'display $hierarchy'. The command 'change_working_pointer #i' sets the working pointer ('$working_pointer') to the context of the instance identified by the value of #i. The command 'file_display DLOCKS_FILE $queues' stores the current contents of all queues of the module instance identified by the '$working_pointer'.

```
macro DLOCKS\
      { file_display DLOCKS_FILE #round_number;\
        file_display DLOCKS_FILE #initial_seed; \
        file_display DLOCKS_FILE $current_firing_step;\
        #i:=0;\
        do\
          #i:=#i+1;\
          if #i > #n_instances then exit\
              else\
                  change_working_pointer  #i;\
                  file_display DLOCKS_FILE $working_pointer;\
                  file_display DLOCKS_FILE $queues\
           fi\
         od\
      }
```

As it was mentioned earlier, we considered the allocation algorithm validated when a long simulation session (with many runs) terminated with all runs ending in a successful deadlock.

5. Conclusions

This paper described the development and validation process of an executable specification of a distributed algorithm solving the most general version of the dynamic resource allocation problem. The target, executable specification is in Estelle - formal description language (technique) for distributed systems. Both the development and validation was done with help of EDT tools. The idea of the paper was to show the problem, to present its distributed, algorithmic solution and to indicate how a working and verified prototype can be quickly obtained if an appropriate language and supporting tools are used. From this point of view the validation by simulation part of the paper is the most instructive, because it shows a scheme to follow in similar, distributed cases. Of course, this specification exercise, though interesting in itself, is not the final goal. Having the prototype, one can experiment, e.g., with the efficiency of the proposed solution in different conditions as if they were real. Such efficiency evaluations are very important with respect to distributed algorithms in general, since mathematical complexity analysis is not very precise and not always very useful for them (worst or average case approximations in number of messages and/or response time). Estelle and EDT tools are especially prepared to perform the efficiency analysis. Such analysis comparing different solutions to the problem of dynamic resource allocation is a subject of a separate study.

References

[Awerbuch and Saks, 1990] Averbuch B., Saks M., A dining philosophers algorithm with polynomial response time, Proc. of the 31st IEEE Symposium on Foundations of Computer Science, 65-74 (1990)

[Budkowski, 1992] Budkowski S., Estelle Development Toolset, J. of Computer Networks and ISDN Systems, vol. 25, No.1 (1992)

[Chandy and Misra, 1984] Chandy K., Misra J., The drinking philosophers problem, ACM Transactions on Programming Languages and Systems 6, 632-646 (1984)

[Choy and Singh, 1993] Choy M., Singh A., Efficient fault-tolerant algorithms for resource allocation in distributed systems, ACM Transactions on Programming Languages and Systems 17, 535-559 (1995)

[Dijkstra, 1971]	Dijkstra E. W., Hierarchical ordering of sequential processes, Acta Informatica 1, 124-149 (1971)
[ISO 9074]	ISO 9074, Information processing systems - Open Systems nterconnection - Estelle - A formal description technique based on an extended state transition model (1989)
[Rhee, 1998]	Rhee I., A modular algorithm for resource allocation, Distributed Computing 11, 157-168 (1998)
[Winkowski, 1998] and	Winkowski J., A Multi-Agent System for Safe Distributed Task Resource Management, Proc. of the workshop 'Multi-Agent Day', ICS PAS Reports, 99-105 (1998)