GALATEA Simulation Language Reference Manual

Galatea Team

Agosto 2023

Contents

Chapter 1

Introducción

1.1 Characteristics of the GALATEA Simulation Language

GALATEA is a multi-agent systems modelling language to be simulated in a DEVS, multi-agent platform.

GALATEA platform allows to model and simulate discrete event, continuous and multi-agent systems. It is the product of two lines of research: simulation languages based on Zeigler's theory of simulation and logic-based agents. There is, in GALATEA, a proposal to integrate, in the same simulation platform, conceptual and concrete tools for multi-agent, distributed, interactive, continuous and discrete event simulation.

DEVs model of the system to be simulated is seen as a set of subsystems, that exchange information in different ways. The subsystems can store, transform, transmit, create, and eliminate information. GALATEA has the following features to represent systems and information processes:

Subsystems are represented by nodes of a network. Nodes may be of different types according to their functions. Information transformation is described and performed by program code associated to each node. Information exchange among nodes is done by the action of the code on shared variables or files, and also by passing messages from a node to other. Information is stored in variables, files or in lists of received messages in the nodes.

The GALATEA Language definition includes declaration and instruction taken from Java []. Also GALATEA can be used as a Java Library.

Java features add the power of a general purpose object oriented language to the modeling and simulation power of GALATEA. Two simple examples follow to give a general idea how GALATEA programs look like.

1.1.1 Example of discrete simulation

Railroad System.

In a simple railroad system, trains, with a number of coaches equal to a random number from 16 to 20, depart from a station each 45 minutes. After 25 minutes of travel, the trains reach their destination. The program writes the time of arrival and the number of coaches. For clarity, in all the manual, GALATEA reserved words are in upper case letters; user identifications are in lower cases with normally the first letter in upper case if identifies a node.

```
TITLE
  Railroad System

NETWORK
  Depart (I) {
    coaches = UNIFI(16,20);
    IT(45);
    SENDTO(Railroad);
  }
  Railroad (R) {
    STAY(25);
    RELEASE SENDTO(Destination);
  }
  Destination (E) {
    WRITE("A train arrives at time: %6.2f Its length is %d\n", TIME, coaches);
    PAUSE;
  }
INIT
  TSIM = 300;
  ACT(Depart, 0);
DECL
  MESSAGES Depart(INT coaches);
  STATISTICS ALLNODES;
END
```

The program has the following sections:

- `TITLE` section with title and explanations.
- `NETWORK` section that describes the nodes (subsystems) and the relations among them.
- `INIT` section that sets the simulation time and the node to be firstly activated at simulation start. It may include other initializations (see Chapter **??**).
- `DECL` section that declares the structure of the messages and the statistics required. It may include declarations of other variables and elements of the program (see Chapter **??**).

It follows a description of the NETWORK section and its functions: After each node name follows a letter between parenthesis indicating its type. `Depart` is a node of `I` (Input) type. When executed, it generates a message and sends it to the node `Railroad`. The message in this example represents a train. It has an integer type field called `coaches`, that indicates the number of coaches. In the code of the node (code inside braces) this field is set to a value given by a GALATEA function `UNIFI` that produces a random integer number, in this case between 16 and 20. A new activation of the `Depart` node is scheduled after an Interval Time (`IT`) of 45 time units.

The node `Railroad` is of R (Resource) type. When executed, it processes the message and it stores it temporarily in an internal list (`IL`). The code sets, by means of the GALATEA instruction `STAY`, that the message must remain in the node for a time equal to 25. When the message is released, it is sent to the node `Destination`.

The node `Destination` is of E (Exit) type. It takes the message, executes the instruction of the code (it writes, for instance: "`A train arrives at 140.  Its length is 17"`) and it deletes the message. See that the value (17) is transported by the message in the variable `coaches`. After writing, the program stops (instruction `PAUSE`) and when the button `Continue` is pressed the simulation continues. New trains are generated in `Depart` each 45 units time, then they pass to `Railroad`, and so on.

The above source program is the input to the compiler system that translates it and runs the simulation. The simulation time is not the real processing time. It takes the value at each event. At the beginning is 0, when the first train arrives is 25, when the second train departs is 45, when the second train arrives it is 70, when the third train departs it is 90, etc.

In GALATEA programs the basic features are implemented in the following way:

- Each node is specified by a name and a node type. The type of the node must be defined. The type implies specific ways of being activated and processing the messages.

- The code can include:

  - Simple GALATEA instructions.
  - Structured GALATEA instructions.
  - Procedures and functions of the GALATEA library.
  - System generated instructions.

- Information exchange among nodes is accomplished by allowing the code of different nodes to share some of the data (global variables, files, fields of the messages). The messages are structured like records. The user can declare different types of `MESSAGE`s with different types of fields. The user's code can use and change the values of the message fields referring to them by its simple name. To deal with different entities sharing common properties, messages of different structure can have some fields with the same name, they correspond to the field name.

  The Messages are used to represent entities traveling from one node to another.

- Information is stored in variables, files and in messages. Messages are stored in two basic lists that are associated with some nodes:

  - EL (Entry List) of received messages to be processed by the node. They may be used to represent queues or waiting lines.
  - IL (Internal List) where messages can be stored until they are extracted. Usually they are used to represent entities using a facility or remaining in delay lines.

The abbreviations `EL` and `IL` will be respectively used for Entry List and Internal List, with the plurals `EL`s and `IL`s.

The execution of the code of a node is called the activation of the node. At a point during the simulation it may be necessary to schedule the activation of certain nodes for the future. The schedule is kept in a list called `FEL` (Future Event List). Each element in the `FEL` represents a pending event. It contains a reference to the node to be activated and the time at which the activation must take place. This list is automatically processed by the system, but the user may also handle it through special GALATEA procedures.

The order of activation of the nodes is essential to the simulation. An event (activation `fact`) begins with the processing of the element of the `FEL`, that has the minimum time value. The node referred by this element is activated (executed) and this starts the event execution.

The code of the node may cause changes in the values of the variables, execution of procedures and message processing. In particular this execution may cause message exchanges. It can also change conditions that allow other nodes to move messages or change values of variables. All this must be done in these events. To perform all these changes it follows a scanning of all the nodes that could be affected by the execution of the node. These scanned nodes are then activated. The scanning is repeated cyclically until no further movements of messages are possible. The event processing is then ended and the following event from the `FEL` will be considered. During the execution of an event the state of the system changes and new future events are scheduled. The value of the simulation time do not change during the transformations produced in the event. When a new event is executed the time variable is updated to the time of the event indicated in the `FEL`. So the simulation process goes on until an end condition specified by the user finishes the simulation run. The user must be aware of the two ways in which a node can be activated: by an event that refers to the node, which starts a new event, or by scanning of the network during an event.

1.1.2 Example of continuous simulation

```
TITLE
  Plantation with periodic cuts.
  In a plantation the quantity of wood growths according to a
  logistic curve whose differential equation is:
                        w' = k * (1 - w / wm) * w
  When the quantity is greater than wm, a cutting process starts
  which decides the proportion of wood to be extracted according
  to a function of the actual price, p(TIME) and the mean price pm.
  The time for the cut process is negligible compared with the
  rates of growth.

NETWORK
  Growth (C) {
    w' = k * (1 - w / wmax) * w;
    if w >= wm ACT(Cut, 0);
    GRAPH("Wood Evolution", TIME, "TIME", w, "Wood");
    GRAPH("Price Evolution", TIME, "TIME", price(TIME), "Price");
  }
  Cut (A) {
    w = w - MIN(0.3 * wm, minCut + MAX(0, cc*(price(TIME) - pm)));
  }
INIT
  TSIM = 50;
  ACT(Growth, 0);
  k = 0.21; wm = 6000; minCut = 500; cc = 40.2; pm = 24;
  w = 100; wmax = 9500;
  DT(Growth, 0.125);
DECL
  DOUBLE k = 0.21;
  DOUBLE wm = 6000;
  DOUBLE wmax, minCut, cc, pm;
  GFUNCTIONS
      SPLINE price() = (0, 30)(10, 20)(20, 15)(30, 25)(40, 30)(50, 42)(55, 45)
END
```

The type C node `Growth` solves the differential equation. When $w \geq wm$ the node `Cut`, of A type, is activated. This reduces the mass of wood by a quantity that is 30% of the `wm` at most, and `minCut` at least. Within these limits it depends on the difference between actual price (`price(TIME)`) and average price `pm` of the wood. The variable `w` must not be declared because it is known to be a state variable of a differential equation.

The time function `price` is given by the 7 pairs of values indicated in the `GFUNCTION` section. For the time 0, its value is 30; for the time 10, it is 20;

etc. The interpolation is made by a `SPLINE` algorithm.

Two graphs are programmed to display, during the run, `w` and the `price` as a function of `TIME`.

In the `INIT` section, values to the parameters and to the initial value of the variable `w` are given. The integration interval `DT` of node `Growth` is also set. The state variables of type C nodes are not declared. All other variables are declared in the `DECL` section.

1.2 Types of nodes

A summary of the characteristics of the predefined node types is shown in the following list. Some nodes can be activated by an event that refers to it. Some may be also activated during the network scanning process. Some node types have `EL` and `IL`, others only `EL`, others none of them. When activation is attempted by the scanning process, it may be inhibited in certain conditions (`EL` empty). Otherwise the code is executed each time that the node is scanned. The user can avoid these repetitions by means of special instructions. All activation of a node starts a scanning process with the exception of type C nodes.

See Chapter **??** for a detailed account of the processing of the different node types.

- I (Input): Generates messages that enter the simulation process. It creates messages and sends them to other nodes. It has neither `EL` nor `IL`. It may only be activated by an event that refers to it. It is never activated again by the scanning of the network.

- G (Gate): Stops or allows message flow. It has an `EL` for the retained messages. It may be activated by an event or during the scanning of the network if the `EL` is not empty. The instruction `STATE`, used to change the state of the gate, is only executed if the node is activated explicitly by an event which refers to the node.

- R (Resource): Simulates resources used by messages. A real value (called the node capacity) is associated to the Resource type node. The messages in the `EL` represent entities that demand a certain quantity of that capacity during a certain period of time. The code has two parts. One examines the incoming messages, the other processes the outgoing messages. The `EL` is examined and, for each message, it is checked if the demanded quantity is available. If it is so, the message is moved to the `IL`, the time of future depart could be scheduled in the `FEL` through `STAY`, and the quantity of resource used for the message is subtracted from the available capacity. If there is not enough capacity, the message remains in the `EL`. When the depart event is executed, the message with the corresponding time of depart is extracted from the `IL` and it is usually sent to a node through the instruction `SENDTO`. The user can control this process of message sending using the RELEASE instruction. Each time the node moves a message, the network is scanned. The part of the code that processes the departing message is only activated by a departing event and executed only once in the event process. The part that deals with the incoming messages in the `EL` may be activated during the scanning of the network, but only if the `EL` is not empty.

- E (Exit): Destroys messages. The message in the `EL` is processed. The code in the node is executed and the message is destroyed. The node is activated during the scanning of the network if the `EL` is not empty.

- C (Continuous): Solves systems of ordinary differential equations of first order. It accepts instructions of the form `<variable>' = <expression>`. The system considers a succession of such instructions as a system of differential equations. When the node is activated, the solving process starts. It may be interrupted and continued when the node is deactivated or activated by special instructions. During the execution the node schedules its own activation at each integration step. This activations does not cause scanning of the network. C type nodes have neither `EL` nor `IL`.

- A (Autonomous): Executes events at scheduled times. It is only activated by an event that refers to it. It is not activated again during the scanning of the network. It has neither `EL` nor `IL`. It can activate itself and other nodes, change variables, and send messages.

1.3 Nodes with indexes

The nodes can have a subscript or index that can take values 0, 1, 2, . . . ; until a maximum value that must be small to that declared by the user in the node heading. Indexed nodes are equivalent to a set of nodes of the same type. The size of the set is equal to the declared value. These nodes share a common code. They are used to represent sets of similar subsystems.

When activated during the scan of the network, they are activated sequentially. When activated by an event that refers to it, only the node with the index of the event is activated. An index variable `INO` takes the value of the index of the currently executed node. The user has access to that index in order to be able to control differences when processing the common code. The multiplicity or dimension of the node is declared as arrays in Java, i.e., in the form `[N]`.

Example:

```
Window[5] (R) {STAY(5); RELEASE SENDTO(Exit);}
```

This line declares that the node named Window is of R type. It consists in 5 nodes that share the code.

The nodes without index are called simple, those with index are called multiple nodes. The number of nodes of a multiple node is called its dimension or multiplicity.

The nodes of a multiple R type node can be of different capacity.

1.4 Example 1: Simple Serving System

The problem is to compute the queues and waiting times of people that are to be served one by one in a window. They enter the system at random and the serving times are also random. After being served they left the system. The GALATEA program may be as follows:

```
TITLE
  Simulation of a simple serving system.

  Customers enter the system to be served at a window.
  The times between arrivals are taken at random from an exponential
  distribution with mean <Tba>. The customers are served or form a
  queue in front of the window. The serving time is taken from a Gaussian
  distribution with mean: <MeanWait> and standard deviation: <StaDev>.
  Once they are served they go to Exit to leave the system.
  The simulation is to go until time 1000. Statistics of nodes are
  required. For clarity reserved words are written in uppercase letters.

NETWORK
  Entry (I) { SENDTO(Window); IT(EXPO(Tba)); }
  Window (R){ STAY(GAUSS(MeanWait, StaDev)); RELEASE SENDTO(Exit); }
  Exit (E) {}

INIT
  TRACE;
  TSIM = 100; ACT(Entry, 0); Tba = 4; MeanWait = 0.8; StaDev = 0.1;

DECL
  DOUBLE Tba, MeanWait, StaDev;
  STATISTICS
    Entry, Window, Exit;
END
```

The program has four sections divided by four separators: `TITLE`, `NETWORK`, `INIT` and `DECL`.

- The first line of the `TITLE` is used to put the title of the model. The rest of the text of the first section can be used to put, author, date, etc. An explanation may be included that can be extended to a full documentation. The only restriction is that no line of the text can start with the words `NETWORK`, `INIT`, or `DECL`; because this is the separator to the next section.

- `NETWORK` section must start with this word as the first in the first line of the section (comments between `/*` and `*/` may however be included at any place in which a blank is allowed). This section contains the nodes of the program. The heading of a node includes its name and the type of the node (a letter between ()). The characters { terminate the heading, after them the code may follows. The code is written in free format. Each statement ends with a semicolon (`;`) and the code statement set ends with a curly brace (}).

The word `INIT` at the beginning of a line indicates the end of the `NETWORK` section. See Chapters **??** and **??** for more details.

The instructions of GALATEA (like `STAY(<expression>)`) are described in Chapter **??**, the procedures (like `ACT`) in Chapter **??**, the functions (like `EXPO` or `GAUSS`), in Chapter **??**. For each user programmed node, the compiler will generate a procedure that contains the user code and system provided code to perform the other operations of the node implied by its type. This procedure is executed when the node is activated.

- `INIT` section contains instructions to give initial values to the variables and parameters for the simulation run. It may contain the specification of the total simulation time by assignation of the value to the system variable `TSIM`. There are other ways to finish the simulation.

  This section must contain at least an ACT procedure to activate one node of the network to start the simulation. In the example the node `Entry`, indicated in the first argument, will be activated at a time equal to the time of the initialization (that is 0 by default) plus the value of the second argument (also 0 in this example). Thus, it is activated at the beginning of the simulation.

  Values are assigned to the auxiliary variables `Tba`, `MeanWait` and `StaDev`. Complete algorithms may be included in this section. In this section may also be assigned: capacities to R nodes, **values to user defined functions, frequency table parameters and database tables REVISAR**. See Chapter **??** for details.

- `DECL` section contains the declaration of some elements of the program introduced by the user. These names must be different of the reserved or system words (see Chapter **??**).

  The subsection `STATISTICS` may contain node names and variable names from which statistics are wanted. In this case only node statistics are requested.

  Other declarations of data corresponding to the base language may be done here (**types, constants, records, arrays, sets, files REVISAR**). The procedures and functions declared and programmed by the user must be also included in this section.

Function defined by pairs of values (a special feature of the GALATEA language), **the data base tables, frequency tables REVISAR** and the structure of the messages must be also declared here. See Chapter **??** for details.

Lexicographic remark

- Galatea reserved words are not case insensitive.
- The translator is case insensitive.
- Comments may be included within parenthesis `/* */` or after `//` until the end of line.

1.4.1 Description of the process

The execution of the program starts with the `INIT` section. The total simulation time is assigned to the system variable `TSIM`. The system variable `TIME`, that keeps the value of the time during the simulation, is initialized to 0. The user could put explicitly other assignation (even a negative number).

The GALATEA procedure `ACT` schedules an activation for the actual time plus the second argument (here also 0, but it may be a real expression) so the first activation will happen at `TIME = 0` in this example. The other instructions of the `INIT` section assign values to some variables.

Then the execution of the `NETWORK` section starts. The components of the simulated system: entrance, window and exit are represented by the nodes `Entry`, `Window`, and `Exit` in the `NETWORK`. The customers are represented by the messages that each node may send to another.

The node `Entry` is of I type. An I type node (or more exactly the class into which the compiler translates it), when activated, generates a message, and sends it to the end of the `EL` of a node through the instruction `SENDTO`. In this node a next arrival event (i.e., the next activation of the node `Entry`) may also be scheduled. This is done by the instruction `IT` (Interval Time). This value is interpreted as the time that has to pass until the next arrival. In this case, this interval is the value computed by the function `EXPO`, that generates a pseudorandom value taken from an exponential distribution (see Chapter **??** for further explanations). So the GALATEA schedules a new activation of `Entry`. This activation will occur at a time that is equal to the actual value of `TIME` plus the value passed to `IT`. Values to the fields of the message can also be assigned in the code of this node.

The node `Window` simulates a resource, in this case the window where the customers are served. The capacity is 1 by default (no other capacity was assigned) so that only one message can enter the `IL`. The messages enter the `EL` of the node. This `EL` represents the queue of customers at the window. The `IL` represents the customer being served. When the node `Window` is activated during the scanning process of the network (and this will happen because the node `Entry` started such a scanning), the procedure of the node type R, if the `IL` is empty, searches the `EL`. If it is empty nothing is further done. If it has messages, the procedure takes the first message to pass it to the `IL`. If the `IL` was occupied (it contains a message representing a customer using the resource) the incoming message remains in the `EL`. When a message is introduced in the `IL`, that means that a new service starts. Then the GALATEA system schedules an event of end of service. That is, a future activation of the node `Window` at a time equal to the actual value of `TIME` plus the value assigned through the instruction `STAY`. In this case the value is taken at random from a gaussian distribution. That time is also kept in a field of the message as its exit time. When that future activation takes place, the node `Window` is again activated, but now by the event that refers to it. In this case, the procedure searches the `IL` for a message having an exit time equal to the time of the event (that is, the actual time). In this simple case only one message may be in the `IL`. The message is then extracted and sent to the `EL` of the node `Exit`. As this message movement triggers a new scanning of the network, the node `Window` is now activated by the scanning process with is `IL` empty. So it searches again to see if there are in the `EL` other message to be passed to the `IL`. If it is so, a storage process as the explained above will happen. Otherwise, the `Window` remains idle.

The node `Exit` is of E type. When it is activated (as a consequence of the scanning of the network started by the extraction event in the node `Window`), the procedure of the node `Exit` removes the message of its `EL` and discards it. During all these operations statistics are recorded of the lists, and the use of the resources.

1.4.2 Trace of the operations

The user can ask, from the model interface, command line or in the code through instruction `TRACE`, for a file with the trace of the simulation. A partial trace of the simulation of this program is shown below. To follow the trace the following remarks must be regarded:

- At the beginning notice that the only event is the arrival introduced in the `INIT` section. The node is `Entry`, its index is 0 (the node is a simple one) an the event parameter is 0.
- The next event is taken from the first element of the Future events List (`FEL`).
- When starting to process each event, the status of the `FEL` is displayed.
- The line: `>>>>>>` indicates the beginning of an event. Its fields: `IEV` (index of the EVent), `TEV` (Time of the EVent) and the node firstly activated in the `NETWORK` are indicated. Then the `FEL` is shown.
- The procedures executed in the activation are shown by means of the `ACT` and `SCAN` labels that indicate the activation or the scan of each node of the network.

In the example the trace displays the activation of the node `Entry` scheduled by the `ACT` instruction in the `INIT` section. This event is an activation of the node `Entry` at `TIME = 0`. The IEV is 1 and the event parameters are TEV equals (`0.000`) and the node where it will occur (`entry[0]`).

The event 1, which activates node `Entry[0]` at time 0, begins by creating a new message identified as (`input0:1`). It then continues to send the message to the entry list (`EL`) of node `Window[0]` which, since it was empty, will have length `EL.ll()` = 1. Finally, the activation ends adding other event (next activation) at `FEL` for future time 0.353 (this number comes from the `EXPO` function). Notice that the messages ID generated in a node contains the name of the node, the index of the node colon a sequential number.

During the network check, when scanning the node `Window[0]`, it is seen that its internal list (`IL`) is empty and that the message `entry0:1` (generated in `Entry[0]`), is in `EL`, so it moves from `EL` to `IL`. At that time, a type 2 event (activation of node `Window[0]`) is placed in `FEL` to schedule the end of service at time 0.876.

The network scan continues, but since nothing happens event 1 ends.

An events of type arrivals happen at times 0.353. It is intercalated in the FEL before the end of service event, scheduled for TIME 0.876. The generated message is sent to `Window[0] EL` and remain there because the `IL` is occupied.

The next event is of type `Window[0]` at TIME 0.876 scheduled at TIME 0. This event examines the `IL` of `Window[0]` seeking for a message with `EXIT TIME = 0.876`. The message is extracted and sent to the `EL` of the `Exit[0]` node. Then the `EL` of `Window[0]` is examined. The next message (`IEV` = 2) in the queue (`Window[0].EL`) is extracted, its exit time is scheduled at TIME 1.743 and the message is moved to the `IL` of `Window[0]`.

In the same event the node `Exit[0]` is scanned, the message `entry0:1` is extracted from its `EL` and discaded.

The network (excluded the `Entry[0]` node that is of I type) is scanned again and, as no messages are moved, the next event in the `FEL` is processed.

The reader can continue the examination of this simulation. In this simple example only two events are at most in the `FEL` (next arrival and next end of service). In more complex models the `FEL` can be longer. The examination of traces is very useful to understand the GALATEA system works. It is also very important for debugging.

```
INIT
 0.000 act (I)entry[0] 0.000
>>>>>1  0.000 entry[0] FEL=[0;0;/]
 0.000 ACT  (I)entry[0]
 0.000 create entry0:1
 0.000 sendto entry0:1 -> window[0] EL.ll()=1
 0.000 act (I)entry[0] 0.353
 0.000 SCAN (E)exit[0]
 0.000 SCAN (R)window[0]
 0.000 EL->IL entry0:1 EL.ll()=0 IL.ll()=1
 0.000 act (R)window[0] 0.876
 0.000 SCAN (E)exit[0]
 0.000 SCAN (R)window[0]
>>>>>2  0.353 entry[0] FEL=[1;1;/] -> (R)window[0], 0.876
 0.353 ACT  (I)entry[0]
 0.353 create entry0:2
 0.353 sendto entry0:2 -> window[0] EL.ll()=1
 0.353 act (I)entry[0] 0.888
 0.353 SCAN (E)exit[0]
 0.353 SCAN (R)window[0]
```

```
>>>>>3  0.876 window[0] FEL=[1;1;/] -> (I)entry[0], 0.888
 0.876 ACT  (R)window[0]
 0.876 sendto entry0:1 -> exit[0] EL.ll()=1
 0.876 SCAN (R)window[0]
 0.876 EL->IL entry0:2 EL.ll()=0 IL.ll()=1
 0.876 act (R)window[0] 1.743
 0.876 SCAN (E)exit[0]
 0.876 left entry0:1
 0.876 SCAN (R)window[0]
 0.876 SCAN (E)exit[0]
 0.876 SCAN (R)window[0]
>>>>>4  0.888 entry[0] FEL=[1;1;/] -> (R)window[0], 1.743
 0.888 ACT  (I)entry[0]
 0.888 create entry0:3
 0.888 sendto entry0:3 -> window[0] EL.ll()=1
 0.888 act (I)entry[0] 1.727
 0.888 SCAN (E)exit[0]
 0.888 SCAN (R)window[0]
```

1.4.3 Standard Statistics

When required by the declaration STATISTICS, the GALATEA produces an output with statistics of nodes and variables. Statistics for one run of the simulation example follows:

```
  Simulation of a simple serving system.
Time:           100.000
Time Stat: 100.000
Replication: 0
Date:           Wed Aug 16 19:23:31 VET 2023
Elapsed time: 0h 0m 3.18s

Input Entries
entry[0] 35

Node.Lst  Ent Len Max Mean Dev MaxSt MeanSt DevSt T.Free
window[0].el 35 0 2  0.059  0.241  0.855  0.169  0.273 93.894
window[0].il 35 1 1  0.269  0.443  0.928  0.789  0.094 72.895
```

```
Exit node Left Mean Dev Max Min
exit[0] 34  0.942  0.262  1.505  0.561

Var/Node.atrb Mean(t) Dev(t) Max Min Mean(v) Dev(v) Actual
window[0].use  0.264  0.441  1.000  0.000  0.667  0.471  1.000
```

The statistics information is stored in the file `Model.sta` that will be found inside the experiment folder `Data_100.0` and inside the replica folder `000`.

The total simulation time and the last time in which the Statistics were cleared (see procedure `CLRSTAT`) are also shown.

If replications of the experiment are asked for, the number of the replication is given.

The date and time of the beginning of the execution and the computing time spent in the simulation are also recorded.

For the I type nodes the number of generated messages is shown.

For each `EL` and `IL` the number of entries, the actual length and the maximum length are displayed. The statistics show the mean length and its deviation, the maximum and mean waiting time of the messages in the list, its deviation and the time that the list remained empty.

For the E type nodes the mean time in the system of the messages discarded at the node and its deviation are recorded.

In this example only the used time of the resources are displayed, because no other variables in the STATISTICS were requested. The statistics given are the mean and deviation of the variables as a function of time, the maximum and minimum values, the mean and deviation of the values and the actual (final) value.

1.5 Example 2. Port with Three Types of Piers

This is the simulation of a port that has three types of piers to receive different types of ships:

- ships with general charge,
- ships that bring charge to be discharged in bulk to silos, and

- ships that come to be repaired.

The port has an entry channel in which only a ship at a time is allowed to pass. The same channel is used to enter to the piers and to exit, once the operation in the port is finished.

```
TITLE
  Model of a Port.
    Ships arrive at a port with interval times with exponential
  distribution with mean Tbarr.
  There are 3 types of ships. The type is selected from an empirical
  random distribution by means of a function FTyp.
  According to the type they go to the piers 0, 1 or 2.
    The ships are queued before the entry channel. A ship is allowed to
  pass only if the channel is free and the ship has a place in the pier
  of its type.
    The time spent in passing the channel is a fixed value TChannel.
  In the pier the ship remains a time taken from a Gamma distribution
  The mean TPier depends on the type. The deviation is 10% of the mean.
  Each ship uses only one position in the pier.
    To the served ships the type 3 is assigned and they are sent to the
  channel to exit the system.
  Make a frequency table of the mean time in the system.
  Experiment with different serving times in the piers.
NETWORK
  Entrance (I) {IT(EXPO(Tbarr)); ShiTyp = FTyp(); SENDTO(Control); }
  Control(G) {
    IF((FREE(Pier[ShiTyp]) > 0) AND (FREE(Channel) > 0))
      SENDTO(Channel);}
  Channel (R) {
    STAY(TChannel);
    RELEASE
      IF (ShiTyp == 3) SENDTO(Departure);
      ELSE SENDTO(Pier[ShiTyp]);
  }
  Pier[3] (R) {
    STAY(GAMMA(TPier[ShiTyp], 0.1 * TPier[ShiTyp]));
    USE = 1; ShiTyp = 3;
    RELEASE SENDTO(Channel);
  }
  Departure (E) {HISTOGRAM("Time in the system", 30, TIME - GT, "Ship");}
INIT
  TSIM = 1200; ACT(Entrance, 0);
  Pier[0] = 4; Pier[1] = 3; Pier[2] = 1;          // Capacities of the piers
  PARAM(Tbarr, 4, "Mean time between arrivals");
  PARAM(TChannel, 0.2, "Time to pass the channel");
  PARAM(TPier[0], 20, "Mean time spent in the pier 0");
  PARAM(TPier[1], 12, "Mean time spent in the pier 1");
  PARAM(TPier[2], 28, "Mean time spent in the pier 2");
DECL
  DOUBLE Tbarr, TChannel;
  INT TPier[3];                                  // Mean time spent in the piers
  MESSAGES Entrance(INT ShiTyp);                    // structure of the messages
  GFUNCTIONS
    FREQ fTyp(INT) = (0, 50)(1, 40)(2, 10);  // Values of frequency of types
  STATISTICS ALLNODES;
END
```

1.5.1 Remarks about the program

The ships are represented by messages generated by the I type node `Entrance[0]`. The messages have a field `ShiTyp`. To declare, that the messages must have this field, a declaration is issued in the subsection `MESSAGES` of the `DECL` section with reference to the generating node.

The `Entrance[0]` node assign value (0, 1 or 2) to the `ShiTyp` parameter of the generated message. The value is given by the function `FTyp` defined in the `GFUNCTIONS` declaration. The values of the frequencies are also given there (value 0 frequency 50, value 1 frequency 40, value 2 frequency 10). The function `FTyp` produces a random number 0, 1, or 2 according to those frequencies. It is assigned to the field `ShiTyp` of the message. The messages are sent to the `EL` of the node `Control[0]`.

In the `Control[0]` node, that represents the control of entry to the port, the GALATEA generated variable `Pier[ShiTyp].FREE` holds at each moment the value of the free capacity of the node `Pier[ShiTyp]`. The maximum capacities are assigned in the `INIT` section: 4 for the `Pier` 0, 3 for the `Pier` 1, 1 for the `Pier` 2. `Channel.FREE` has the same meaning for the resource Channel. In this case it is 1 if the channel is free and 0 if it is engaged. When the `Control` node, that is a type G node, scans the `EL` of messages and executes the code for each message. Only if it find a message for which is true the condition of the `IF` (there is free capacity in the `Pier` of its type and the `Channel` is free), the `SENDTO` instruction is executed. This extracts the message from the `EL` and sends it to the `EL` of the node `Channel`.

The channel is represented by the resource type node `Channel`. When activated by the scanning of the network, this node takes the message in its `EL` and passes it to the `IL` (this will be empty because the Control only allows to pass a message if `Channel.FREE` = 1, that means that no message is using the resource. The exit from the channel is scheduled for a later time given by the user defined variable `TChannel`. When the message abandons the resource the RELEASE instruction that takes care of the outgoing message. In the instruction associated to the RELEASE the message is disposed according to its type. If it is 0, 1, or 2 it is send respectively to the `Pier[0]`, `Pier[1]`, or `Pier[2]`. If it has type 3 (that is the type that will be assigned to the ships abandoning the piers) it is sent to the `Departure` node.

The node `Pier` is a multiple node of dimension 3. Each node represents a different pier; each may receive the corresponding type of ship. The capacity of each node is assigned in the `INIT` section as said above. The user can define the quantity of resource used by each message. This is done by the instruction `USE = 1`, i.e., it is assumed that all the ships use the same space, equal to 1 unit of the resource. This would be different with ship of very different size.

The operation time in the pier is taken from a Gamma distribution; for different types of ships the mean value used is different. The values of the means are assigned in the `INIT` section. Notice that the type of the received ship is put to 3. After the operation it is sent to `Channel` and after passing the channel, the ship of type 3 is sent to `Departure`.

The `Departure` node is of E type. It takes each message of its `EL`, executes the code and discards the message. The code is the computation of the time spent in the system. The `GT` is a GALATEA defined field of the message that keeps its generation time. As `TIME` is the value of the actual time, `TIME - GT` is the total time that the message remained in the system. The procedure `HISTOGRAM` put this value in the frequency table titled `"Time in the system"`. In this case the table has 30 intervals. The table is saved in the file ”Time_in_the_system.dat” and a histogram is shown when the model runs.

The instruction `INTI` in the `INIT` section allows for interactive change of the values of tbarr and `tChannel` for different experiments.

1.5.2 Standard Statistics

```
  Model of a Port.
Time: 1200.000
Time Stat: 1200.000
Replication: 0
Date: Thu Aug 17 17:35:52 VET 2023
Elapsed time: 0h 0m 2.147s

Input Entries
entrance[0] 296

Node.Lst  Ent Len Max Mean Dev MaxSt MeanSt DevSt T.Free
control[0].el 296 1 5  0.337  0.745 26.389  1.317  3.403 930.247
pier[0].el 140 0 1  0.022  0.147 12.241  0.188  1.188 1171.808
pier[0].il 140 4 4  2.267  1.147 24.633 19.785  1.798 75.380
pier[1].el 135 0 1  0.008  0.088  3.219  0.069  0.400 1188.754
pier[1].il 135 1 3  1.335  1.065 16.382 11.935  1.181 310.212
pier[2].el 20 0 1  0.023  0.151 27.889  1.394  6.078 1170.227
pier[2].il 20 1 1  0.456  0.498 32.961 28.098  2.490 651.938
channel[0].el 584 0 2  0.013  0.121  0.400  0.026  0.071 1184.401
channel[0].il 584 0 1  0.097  0.297  0.200  0.200 NaN 1081.316

Exit node Left Mean Dev Max Min
departure[0] 289 18.558  7.119 68.569  9.758

Var/Node.atrb Mean(t) Dev(t) Max Min Mean(v) Dev(v) Actual
pier[0].use  2.267  1.147  4.000  0.000  2.240  1.126  4.000
pier[1].use  1.335  1.065  3.000  0.000  1.401  1.025  0.000
pier[2].use  0.456  0.498  1.000  0.000  0.399  0.490  1.000
channel[0].use  0.097  0.296  1.000  0.000  0.549  0.498  1.000

=== HISTOGRAMS ============================
Histogram: Time in the system of Ship
Intervals Freq. Rel.Fr.
```

```
[9.758;11.718) 35  0.121 ***********************
[11.718;13.679) 60  0.208 ****************************************
[13.679;15.639) 15  0.052 **********
[15.639;17.599) 21  0.073 **************
[17.599;19.560) 36  0.125 ************************
[19.560;21.520) 53  0.183 ***********************************
[21.520;23.480) 29  0.100 *******************
[23.480;25.441) 9  0.031 ******
[25.441;27.401) 5  0.017 ***
[27.401;29.362) 8  0.028 *****
[29.362;31.322) 3  0.010 **
[31.322;33.282) 4  0.014 **
[33.282;35.243) 4  0.014 **
[35.243;37.203) 2  0.007 *
[37.203;39.163) 2  0.007 *
[39.163;41.124) 0  0.000
[41.124;43.084) 0  0.000
[43.084;45.045) 0  0.000
[45.045;47.005) 1  0.003
[47.005;48.965) 0  0.000
[48.965;50.926) 0  0.000
[50.926;52.886) 0  0.000
[52.886;54.847) 0  0.000
[54.847;56.807) 1  0.003
[56.807;58.767) 0  0.000
[58.767;60.728) 0  0.000
[60.728;62.688) 0  0.000
[62.688;64.648) 0  0.000
[64.648;66.609) 0  0.000
[66.609;68.569] 1  0.003
Total 289  1.000

============================================
```

1.6 Compilation Details

Although compiling details are transparent to users some information may be useful.

The source program is written in a file. It is read and processed by the GALATEA compiler that translates it to several Java classes. The program code of each node is translated into a class.

The program so produced by the GALATEA compiler may be then compiled and executed by a Java Development Kit.

Chapter 2

Declatarions

A GALATEA model uses values of different types. GALATEA includes definitions for certain types, variables, constants and functions required for general processing. Others are also system defined to meet special requirements of particular models.

Besides, users may define types, variables, and functions needed to program their models. Other instances of language elements may also be defined: messages, functions given by pairs of values, frequency tables and needed statistics.

All user variables must be explicitly declared except for C-type node state variables.

The names assigned to types, variables and functions are called identifiers. They are strings of letters, numbers and the sign _ . The first character in the identifier must be a letter. GALATEA is case sensitive, so `WaterTemp` and `WATERtemp` are different variables. **REVISAR Esto del case sensitive o insensitive no está muy claro en el compilador**

A declared identifier must not duplicate any system or previously user defined identifier. See Chapter **??** for a list of system defined identifiers.

The declaration section begins with the separator `DECL` as the first word of a line. Identifiers are declared according to their element class.

2.1 SIMPLE TYPES

The following list shows the set of simple types in GALATEA.

- BYTE: Represents an 8-bit signed data type. So you can store numeric values from -128 to 127 (inclusive).
- SHORT: Represents a 16-bit signed data type. This way it stores numeric values from -32768 to 32767.
- INT: It is a 32-bit signed data type for storing numeric integer values between -2.147.483.648 and 2.147.483.647.
- LONG: It is a 64-bit signed data type that stores numeric integer values between -2^{63} to $2^{63}-1$ (-9×10^{18} to 9×10^{18} approx.).
- FLOAT: It is a data type for storing 32-bit single-precision floating-point numbers.
- REAL or DOUBLE: It is a data type for storing 64-bit double-precision floating-point numbers.
- BOOLEAN: It is used to define boolean data types. That is, those that have a value of true or false.
- CHAR: It is a data type that represents a single 16-bit Unicode character.
- STRING: It is used to define a succession of alphanumeric, punctuation marks, blank spaces or any other 16-bit Unicode character.

2.2 TYPE

A type declaration specifies that a given identifier or name will designate a type of data that may be referred to in the model, i.e., an abstract set of data values. The identifier represents a group of constants (unchangeable values).

Type identifiers may be referred to in other type declarations. New types may be defined for convenience of clarity and maintenance. The declaration syntax is:

`TYPE <identifier> {<list of constants>};`

`<identifier>` is the name of the type.

`<list of constants>` is a comma-separated list of the possible values that a variable of this type can take.

Example:

```
  ....................................
  DECL
    TYPE Opcl = {Close, Open};
    Opcl LineGate;
  .....................................
```

Se declara un nuevo tipo de dato `Opcl` cuyos posibles valores son `Close` y `Open`. La variable `LineGate` se declara de tipo `Opcl` por lo que solamente puede tomar los valores `Close` y `Open`.

2.3 MESSAGES

In this section the user may define structures of messages with user's defined fields, besides the default fields (see ??). The declarations must have the heading `MESSAGES`. After this a list of one or more message declarations follow. The declaration syntax is:

`| <node identifier> | <message identifier> | (<list of declaration>)`

`<node identifier>` is the name of a I type node. Index must not be included in this declaration. The declaration defines the structure of the messages generated at that Input type node.

`<message identifier>` is an identifier that may be used by a `CREATE` instruction to generate messages with the declared structure. The `CREATE` instruction may be used in any node, except I type nodes

`<list of declaration>` is a list of field declarations with the same syntax that a variable declaration.

Example:

```
  ShipArr (I){
    shipType = FTypeFreq();
    IT(EXPO(Tba[shipType]);
    ..................................
  }
  TrainContr (A) {
    ..................................
    CREATE(train){
      wCars = ROUND(TotalLoad / Capacity);
      SENDTO(Yard)
    }
```

```
}
......................................
DECL
  TYPE St = (Freighter, Tanker, Barge)
  MESSAGES
    ShipArr(St shipType, DOUBLE load[5], DOUBLE displ),
    train(INT wCars, DOUBLE Capacity);
.......................................
```

In the node `ShipArr`, messages representing different types of simulated ships are generated. The characteristics (fields of the message) are: `shipType`, five quantities indicating different kinds of `loads`, and `displ`acement. In the node `TrainContr`, according to conditions computed by a complex code, the decision of sending a message (`train`) to a `Yard` is taken.

2.4 GFUNCTIONS

GALATEA allows to define functions of one variable given by a set of pairs of values (points). The first element of the pair is the value of the argument (it belongs to the domain). The second is the corresponding value of the function.

There are two types of GFUNCTIONS,

1. Interpolation functions declared as follows:

   `<interpolation method> <function identifier> () = <list of points>;`

   `<interpolation method>` indicates the method to be used to compute values for intermediate values of arguments not given in the set of pairs. The methods may be:

   - `STAIR`: the function value corresponding to an intermediate value of the argument is the corresponding to the more near value of the arguments in the set of values lesser than the given argument, i.e., the function is supposed piecewise constant and continuous from the right.
   - `POLYG`: a linear interpolation is used.
   - `SPLINE`: a cubic spline interpolation is used.

The types of the argument and of the function are not declared because always both are real.

2. No interpolation functions declared as follows:

   - `DISC`: The function is only defined for the values given in the set of pairs. Intermediate values are not allowed.
     Syntax:
     `<function type> DISC <function identifier> (<argument type>) = <list of points>;`
   - `FREQ`: The value of the function is taken at random among the possible values of the first element of each pair. The second element is interpreted as proportional to the probability of the corresponding first value.
     Syntax:
     `FREQ <function identifier> (<argument type>) = <list of points>;`

The types of the first element of the points (`<argument type>`) and of the second (`<function type>`) may conform with the following table:

Method	Argument	Value
DISC	enumerate or real	enumerate or real
STAIR	real	real
POLYG	real	real
SPLINE	real	real
FREQ	enumerate or real	real (≥ 0)

Example:

```
............................
TYPE TTS   {FREIGHTER,BULK,TOURIST,REPAIR};
TYPE TExit {A,B,C};
TYPE TTT   {STOPPING,VISIT,TRANSFER};
GFUNCTIONS
  STAIR TFreig()   = (0.0,7)(0.2,13)(0.4,15)(0.6,19)(0.8,21)(1.0,22),
  POLYG TRepair()   = (0.0,55)(0.3,72)(0.7,81)(0.9,88)(1.0,90),
  SPLINE TBulk()    = (0.0,9)(0.2,12)(0.4,16)(0.6,19.5)(0.8,22)(1.0,22.5),
  TExit DISC FClaExit(TTS) = (FREIGHTER,A)(BULK,A)(TOURIST,B)(REPAIR,C),
  FREQ FType(TTS)   = (FREIGHTER,611)(BULK,423)(TOURIST,212)(REPAIR,15);
..............................
```

The function declaration can be done without assigning the list of points. In this case, the points can be assigned to the function in the INIT section (See **??**) using the PARAM statement (See **??**).

2.5 STATISTICS

The GALATEA system collects statistics of nodes and variables. The nodes and variables for which statistics are desired must be specified by the user. After the heading `STATISTICS`, a list of node identifiers and simple variable identifiers may follow.

The variables for which statistics are required, must be initialized in `INIT`.

If one of the elements of the list is the reserved word `ALLNODES`, the statistics of all nodes are given.

The statistics are always displayed at the end of the simulation run, but they may also be called by the `STAT` procedure in the program or by user's interruption during the execution of the program.

Examples:

1. `STATISTICS Dock, Crane, TotalWeight;`

   Statistics of the nodes `Dock` and `Crane`, and the variable `TotalWeight` will be displayed.

2. `STATISTICS ALLNODES, TotalCash, Money[1], Money[2], NumberTrans;`

   Statistics of all the nodes, and the variables `TotalCash`, `Money[1]`, `Money[2]`, and `NumberTrans` will be displayed.

Chapter 3

Inicializations

In the INIT section of a GALATEA model the user puts the instructions to give the particular values to the data to be used in a simulation run.

3.1 Initial Values to Simple Variables, R-Nodes capacities and Arrays

For simple or indexed variables assignative instructions are used;
Examples:

```
pressure = 15; temp = 275.0; r = 8.2056E-2; n = 4; ch = 'Y';
volume = n * R * temp / pressure;
FOR(i=1; i<=7; i++) concentr[i] = fConc();
FOR(INT j=0;j<6;j++) Room[j]=10;          // Capacity of each Room
```

`fConc` may be a user's defined function, a `FREQ` type function, etc.
`Room` is a multiple R-type node. The capacity of each `Room` is set to 10.
For arrays may be used:
Examples:

```
INT f[4] = {0, 0, 0, 0};
DOUBLE MI[TTypTur] = {5, 27, 72.0};
INT table[2][3] = {{0, 0, 0}, {0, 0, 0}};
```

`TTypTur` must be a user's defined `TYPE`, and `table` is an array with 2 rows and 3 columns.

3.2 Initial Actions

At least one node must be activated in the `INIT` section in order to start the simulation. This is done by an ACT instruction (See **??**).

Instructions like:

- `TRACE`, to start the tracing (See **??**),
- `OUTHEADER`, to set the header an output file (See **??**),
- `DT`, to set the integration step that will be used in the indicated C type node (See **??**),
- `METHOD`, to set the integration method in a C type node (See **??**),

are often used in the `INIT` section.

3.3 Parameters

Using the `PARAM` instruction, each variable that is a model parameter is set to a default value and a description (see **??**). The `PARAM` instruction can also be used to return model parameters to GFUNCTIONS points (See **??**).

3.4 Interactive Experiments

Chapter 4

System predefined nodes

The GALATEA system provides different types of nodes that differ in the activation and message processing. They are summarized in the Introduction. This chapter gives a detailed description and examples. In the following we refer to GALATEA common instructions and procedures to instructions and procedures that can be used in any type of node. In general are instructions or procedures whose action does not depend on the message being processed. They are:

ACT, BEGINSCAN, BLOCK, ASSI, CLRSTAT, DEACT, DEBLOCK, DBUPDATE, DOEVENT, DONODE, ENDSIMUL, EXTFEL, EXTR, FILE, FREE, GRAPH, INTI, IT, LOAD, MENU, NT, OUTG, PAUSE, PUTFEL, REL, REPORT, SCAN, SORT, STAT, STOPSCAN, TAB, TITLE, TRACE, TRANS, TSIM, UNLOAD, UNTRACE, UPDATE, USE.

BEGINSCAN and STOPSCAN are included here, because they can be used in any nodes as part of the associate instruction of SCAN.

The admissible GALATEA instructions, functions and procedures, are listed in chapters 6, 7 and 8.

4.1 I (Input) type nodes

This type of nodes are used to simulate entrance of items to the system. An I type node creates messages and sends them to other nodes through a `SENDTO` instruction. It has neither EL nor IL.

4.1.1 Code

It may contain GALATEA common instructions and procedures, GALATEA functions and the GALATEA instructions SENDTO and COPYMESS.

4.1.2 Activation

It may be activated only by an event that points to the node, and only during the first scanning of the network. If it has an index, only the node with the index of the event is activated.

4.1.3 Function

When activated:

1. A message is generated that always contains the default fields:

   - `GT` Generation Time: contains the value of `TIME` at the generation.
   - `USE` quantity of resource to be used in node R. Initialized to 1 as default value.
   - `ET` to put the time of entrance in a list (`EL` or `IL`).
   - `XT` to put the time of future exit from a list (`EL` or `IL`).
   - `NUMBER` number of the message generated in the node.
   - `NODE` name of the node.

   The user can change the `USE` by a `USE` instruction. `NUMBER`, `NODE` and `GT` remain constant. `ET` are changed by the GALATEA system each time the message enters a list. `XT` is updated each time the message enters a `IL` and a departure time from it is scheduled.

2. If there is an `IT` instruction, a next activation of the node is scheduled to happen at the time `TIME + IT`.

3. The user's instructions are processed. These may usually be assignation of values to the message fields and `SENDTO` instructions to the message to other nodes (one and only one `SENDTO` is to be executed for each message). Assignation of values to the fields are made by assigning values to the corresponding field variables.

4.1.4 Relation with other nodes

The node must send the created message to a node that can accept messages in their `EL`, but it must not receive message from others nodes.

4.1.5 Indexes

In a multiple node the activations of the nodes occur independently.

4.1.6 Examples

1. `EntradaSimple (I) {SENDTO(OtherNode);}`

   The node `EntradaSimple` can only be activated from the `INIT` section or from another node. It generates a message and sends it to `OtherNode` node.

2.
   ```
   MainEntrance (I){
     IF(Number < 40) IT = TRIA(TBAMin, TBAMode, TBAMax);
     ClassOff = UNIFI(1,2);
     IF(ClassOff == 1) SENDTO(Office1)
     ELSE SENDTO(Office2);
   }
   ```

   Messages are generated and sent to `Office1` or `Office2` according to the field `ClassOff`, whose value is given by the integer uniform distribution `UNIFI(1,2)`. Up to 40 messages are generated. The time between successive generations are taken from a triangular distribution `TRIA`.

3.
   ```
   Samples[7] (I) {
   IT(EXPO(MeanArrT[INO]));
   BactConc = GAMMA(MBac[INO], 0.2 * MBac[INO]);
   SENDTO(LabSec[INO]);
   }
   ```

   The messages (samples to be analyzed in a Lab) arrive at random from seven original points and after being assigned a value for BactConc by means of a GAMMA function, they are delivered to one of the seven nodes LabSec (according to their source).

4.
```
Hangar (I) {
  SENDTO(City);
  COPYMESS(5)
    IF(ICOPY <= 3) SENDTO(Quarry[ICOPY]);
    ELSE SENDTO(Mine);
}
```

   The produced message (Truck) is sent to `City`. Five copies are made. The 0, 1, and 2 are sent respectively to `Quarry[0]`, `Quarry[1]`, and `Quarry[2]`. The other two (3 and 4) are sent to `Mine`.

4.2 G (Gate) type nodes

This type of node retains or allows message flow. The nodes type G have an `EL` for the retained messages.

4.2.1 Code

It may contain GALATEA common instructions and procedures, GALATEA functions, the GALATEA instructions `ASSEMBLE`, `COPYMESS`, `CREATE`, `DEASSEMBLE`, `SYNCHRONIZE`, `SENDTO`, `STATE` and the procedures `STOPSCAN` and `BEGINSCAN`.

4.2.2 Activation

It is activated by an event or during the scanning of the network. The instruction `STATE`, used to change the state of the gate, is only executed if the node is activated by an event which refers to the node.

4.2.3 Function

1. When the node is activated by an event that refers to it and there is an instruction `STATE`, this is first executed. After this execution or when the node is activated because the scanning of the network, the other (non `STATE`) part of the code may be executed.

2. If the `EL` is not empty, it is scanned starting from the first message. Each message is examined, and the user code is executed. After this execution, the scanning continues with the next message.

3. If in the above process a `SENDTO` instruction is executed, the message is extracted and sent to the indicated list or node. The scanning of the `EL` continues.

4. If a `STOPSCAN` procedure is executed, the scanning is stopped and the process of the node finishes. However, it may be re-started during the same event if the scan of the network activates again the node. `STOPSCAN` is used, for instance, if the sending of further messages depends on the changes caused in other nodes by the actually sent message.

5. If a `BEGINSCAN` procedure is executed, the scanning begins again from the first element. This may be useful if the sending of a message change the sending conditions of the already examined messages. Unwise use of this procedure may cause a loop.

6. Note that to send a message a `SENDTO` must be executed one and only one time for the examined message.

4.2.4 Relation with other nodes

A G type node must receive and send messages to other nodes.

4.2.5 Indexes

A multiple G type node has as many `EL` as its multiplicity. When it is activated all the nodes are executed sequentially. The variable `INO` has the number of the node being executed.

4.2.6 Examples

1. `WaitingRoom (G) {}`

   When activated, this node scans the `EL` and nothing is done. From other nodes it is possible to extract and modify the messages of its `EL`.

2.
   ```
   Inspect (G){
     IF(Defect){
       STOPSCAN;
       SENDTO(Repair);
   ```

```
  }
  ELSE SENDTO(Sale);
}
```

If the boolean field `Defect` of the message (article) is `TRUE`, the examination of the list stops and the message is sent to `Repair`. Otherwise, it is sent to `Sale`. In the first case, the scan is stopped (will continue in the next activation during the same event); in the second case, the scan continues. Note that the `STOPSCAN` must be used only if it is necessary (see next example) because it increases the processing time by repeating the scanning of the network.

3.
   ```
   Semaf (G) {
     STATE{
       greenLight = !greenLight;
       IT = 25;
     }
     IF((LL(Street.IL) < 20) && greenLight) SENDTO(Street);
     STOPSCAN;
   }
   ```

   Each 25 units of time the boolean variable `greenLight` changes its true value. The node allows the messages (cars) to pass if `greenLight` is true and the `IL` of the node `Street` has less than 20 messages. When a message passes, the scanning of the `EL` is stopped to allow the change in the `IL` of `Street`. Without the `STOPSCAN` more than 20 messages would be sent to `Street`. On the other hand, if the condition is not fulfilled stopping of the scan avoids unnecessary examination of the `EL`. Eventually, future changes in the `IL` of `Street` will produce a network scanning an a new activation of `Semaf`

4.
   ```
   Selection (G){
   IF((Typ == Special) && theFirst){
       SENDTO(Depot);
       BEGINSCAN;
       theFirst = FALSE;
     }
     ELSE IF(!theFirst) SENDTO(Depot);
   }
   ```

Among the messages in the EL of Selection there is one and only one with the field Typ = Special. Only after it passes, all the others may pass. The boolean variable First is originally TRUE.

4.3 R (Resource) type nodes

This type of nodes simulates resources used by the entities represented by the messages. A real value, called the node capacity, is associated to the node. The messages in the EL represent entities that demand a certain quantity of that capacity during a certain time. The EL is examined and, for each message, it is checked if the demanded quantity is available. If it is so, the message is moved to the IL, the time of future depart is scheduled in the FEL, and the quantity of resource used for the message is subtracted to the available capacity. If there is not enough capacity, the message remains in the EL. When the departing event is executed, the message is removed from the IL and must bi send to a node. So, the node has two function an accepting function that assigns resource to a message and put it in the IL and a departing function that extracts the message of the IL and frees resource.

4.3.1 Code

The type R node may contain GALATEA common instructions and procedures, GALATEA functions, the GALATEA instructions PREEMPTION, RELEASE, STAY. If there is an instruction RELEASE (only one is allowed) its associated instruction may contain GALATEA common instructions and procedures and GALATEA functions, the instructions SENDTO, DEASSEMBLE, EXTR, CREATE, and COPYMESS, and the procedures NOTFREE, BLOCK and DEBLOCK.

4.3.2 Activation

The part of the code that processes the departing message (RELEASE instruction) is only activated by a departing event that refers to the node. This event can be only introduced in the FEL by the node itself when a STAY instruction is executed. The node must not be activated by an ACT instruction. The part that controls the entrance of the messages to the IL is activated during the scanning of the network.

4.3.3 Function

1. The user can control the departing process of the message by means of a `RELEASE` instruction. This is only executed if the R type node is activated by an event that refers to the node. The associated instruction is executed only one time at the beginning of the event and not repeated during the scanning of the network. If it is found, the message is extracted and the user's code in the associated instruction is executed. This code must have a `SENDTO` instruction to send the message to a node.

2. After the sending of the message the capacity of the node is updated adding the value of the `USE` field of the message to the free capacity (`<node>.FREE`) and subtracting the value of USE to the used capacity (`<node>.USE`). This resource freeing can be inhibited if a `NOTFREE` procedure appears in the associated instruction of the `RELEASE` instruction. This is used in applications in which some items abandon the resource, but this requires an additional time to be available again. The freeing is made elsewhere by a FREE procedure at a latter time.

3. The part of the code that passes the messages to the `EL` (accepting function) examines the messages in the `EL` starting from the first. For each examined message the user's instructions in this part are executed. One of them may be a `STAY = <expression>` instruction. The required amount of resource (`USE` field) is compared with the available resource (`<node>.FREE` variable). If `USE` is greater than `<node>.FREE`, the message remains in the `EL` and the scanning of the `EL` continues. Otherwise, the message is extracted, the event of its future departing is scheduled (if there was an `STAY` instruction) and the message is added at the end of the `IL`.

   The capacity of the R node is updated subtracting the value of the `USE` field of the message to the free capacity (variable `<node>.FREE`) and adding the value of `USE` to the used capacity (variable `<node>.USE`). The scan of the `EL` continues to look for other candidates to use the resource. If there is not `STAY` instruction and one message enters the `IL`, no departing event is generated. The message will remain in the `IL` until it is extracted by an instruction `REL` (see 6.18), that refers to the `IL` of the node. It extracts the first message without assigned exit time; it updates the quantity of free and used resource and it sends the message to the desired node.

4. If there is the instruction PREEMPTION (see 6.17), some of the above operations are modified.

5. If the capacity of the node was defined as `MAXREAL`, it is assumed to be infinite and no update is made in the free or used capacity.

4.3.4 Relation with other nodes

A R type node must receive and send messages to other nodes.

4.3.5 Indexes

Multiple nodes are processed successively, when the node is activated during the scanning of the network. The `INO` variable takes the value of the index of the processed node. If the multiple node is activated by a departing event, only the departing part (system produced or `RELEASE`) of the node with the index of the event is processed.

4.3.6 Examples

1.
```
Disposal (R) {}
--------------
INIT  Disposal = MAXREAL;
```
   The node `Disposal` enters any quantity of messages (compatible with memory available) in its `IL`. They may be extracted by a `REL` instruction in other node.

2.
```
content...Crane (R) {
  STAY = GAMMA(travel_T, dev_T);
  RELEASE{
    transported = TRUE;
    NOTFREE;
    SENDTO(Base);
  }
}
```

If the `Crane` is free (capacity 1 is assumed), a message (box) takes it; it uses the resource for a time took from a `GAMMA` distribution. When this time is over, the boolean field `transported` is put to `TRUE` and the message is sent to `Base` without freeing the resource. This must be freed by a `REL` instruction.

3.
```
Sea (R) { STAY = FTravelTime(VesselType); }
----------------------------------------
INIT Sea = MAXREAL;
```

   The messages (vessels) enter the resource `Sea` of infinite capacity and remain there for a time given by a function `FTravelTime` that depends on `VesselType`.

4.4 D (Decision) type nodes

4.4.1 Code

4.4.2 Activation

4.4.3 Function

4.4.4 Relation with other nodes

4.4.5 Indexes

4.4.6 Examples

4.5 E (Exit) type nodes

These nodes discard messages. The messages in the `EL` are processed. The code in the node is executed and the messages are discarded. The node is activated during the scanning of the network if the EL is not empty.

4.5.1 Code

It may contain GALATEA common instructions and procedures, GALATEA functions, and the GALATEA instructions `SENDTO`, `CREATE` and `DEASSEMBLE`,

4.5.2 Activation

It is activated by an event that refers to it or during the scanning of the network, if the `EL` is not empty.

4.5.3 Function

1. The `EL` is scanned from the beginning to the end.

2. For each examined message the code is executed.

3. If there is a `SENDTO` instruction, the fields of the message are updated and the message is sent to other node. Otherwise, the message is discarded and the used memory may be re-used.

4.5.4 Relation with other nodes

An E type node must receive and send messages to others nodes.

4.5.5 Indexes

If a multiple E type node is activated the nodes are executed successively. The value of the variable `INO` corresponds to the node being processed.

4.5.6 Examples

1. `Exit (E) {}`

   The messages arriving at E are destroyed.

2.
   ```
   Departure (E) {
     HISTOGRAM("Time in the system", 30, TIME - GT, "Ship");
   }
   ```

   The time that the messages remained in the system is computed by subtracting from the actual `TIME` the generation time `GT`. This value is recorded in a frequency table titled "Time in the system".

4.6 C (Continuous) type of nodes

These nodes solve systems of ordinary differential equations of first order. These nodes do not have `EL` or `IL`.

4.6.1 Code

It may contain GALATEA common instructions and procedures, GALATEA functions, the GALATEA instruction `CREATE`, and the procedures `RETARD` and `METHOD` (see 7.18 and 7.13). It accepts successions of instructions of the form `<variable>' = <expression>`. These may be mixed with other type of instructions. In different C type nodes the METHOD and path of integration may be different. In the above instruction `<variable>` is a state variable that should not be declared. It may have a numerical index. If the variable `TIME` appears in some equation, it must be represented by this name. If the value is transferred to other variable, then both will differ without knowledge of the user, because the solving algorithm changes the variable `TIME` without updating the model variable.

4.6.2 Activation

The node is first activated from the `INIT` section or from other node. Then the solving process starts. During the execution the node activates itself at each integration step. This activations does not produce scanning of the network. They may be interrupted and continued when the node is deactivated or activated by the instructions `ACT` and `DEACT`.

4.6.3 Function

1. In a C type node one or more systems of differential equations are solved. The whole code is executed at each integration path and the execution of the events of all the program interleaved with those of the differential equations. On the other hand, the C node can generate events and messages by conditions that may depend on the values computed for the differential equations. Thus, a symbiosis of continuous and discrete simulation is complete.

2. As the self-activations of the node do not produce scanning of the network, all actions in the network that depend on the values computed by the differential equations must be explicitly programmed in the node (see examples below). The C type node can be called to be executed at any time from any other node. The calling instruction is:

   ```
   <name of the C node > (0);
   ```

   When this happens the node solves the systems for the actual value of the time and then the control goes back to the calling code. So an updated value of the computed variables can be used by the calling code. The C type node continues solving the system at the originally prescribed intervals.

4.6.4 Relation with other nodes

The node does not receive messages but it can have send messages. As was explained above, the node can be called from any node for an instantaneous evaluation of the continuous variables.

4.6.5 Indexes

4.6.6 Continuous Variables Declaration

4.6.7 Examples

4.7 A (Autonomous) type nodes

These nodes execute their code at scheduled times. They have neither `EL` nor `IL`. They may activate themselves and other nodes, change variables and send messages.

4.7.1 Code

It may contain GALATEA common instructions and procedures, GALATEA functions, and the GALATEA instructions `CREATE` and `SENDTO`. The common instruction `UPDATE` can only be used in the associated instruction of `CREATE` or `SENDTO`.

4.7.2 Activation

An A type node is only activated by an event that refers to it. It is not activated again during the scanning of the network. If the node is multiple, only the node with index value equal to the index of the event is activated.

4.7.3 Function

When the node is activated the code is executed.

4.7.4 Relation with other nodes

Messages cannot be sent to it, but if messages are created on it these messages must be sent to other nodes.

4.7.5 Indexes

A multiple type A node is a set of independent nodes that can be activated independently.

4.7.6 Examples

1. `Print (A) {IT = 365; WRITE("VALUE: %d\n", x); }`

   Each 365 unit times the writes the the value of `x`. The node must be activated for the first time from other node or from the `INIT` section.

2.
```
Results (A) {
  IT(0.125);
  GRAPH("Prey predator model",TIME,"t",r,"Rabbits",f,"FOXES");
}
```

   Each interval time of 0.125 units (8 times per year) points with the new values of `r` and `f` are added to the time graphic and united by a line to the previous point. The node must be activated for the first time from other nodes or from the `INIT` section.

3.
```
NewStr (A){
  IT = 0.5;
  If(CondStCh){
    ACT(NewSource, 0);
    CREATE(MessSource)
    SENDTO(NewSource);
    DEACT(Inst3);
    ACT(Proc4, 12);
    STAT;
    CLRSTAT;
  }
}
```

Each 0.5 units of time the boolean variable `condStCh` (that may compute conditions for a structural change) is evaluated. If it is `TRUE`, the nodes `NewSource` and, with a delay, `Proc4` are activated. The `Inst3` is deactivated. A message is sent to the newly activated node. The statistics are displayed and then initialized for a new statistical gathering.

4.8 General type nodes

4.8.1 Code

4.8.2 Activation

4.8.3 Function

4.8.4 Relation with other nodes

4.8.5 Indexes

Chapter 5

Network

Chapter 6

Instructions

6.1 ACT schedules a future activation of a node

Syntax:

```
ACT(<node name>, <real expression>);
```

It is used to schedule a node activation in a future time.

`<node name>` is the name of the node to be activated.
`<real expression>` is a real expression whose value indicates the time from now until the future activation.

When this instruction is executed, a future event is put in the `FEL`. The event node is the indicated by `<node name>`. The time of the event is `TIME+<real expression>`. Here `TIME` is the system variable that contains the value of the actual simulation time. The index and the event parameter of the event are the actual values of the variables `INO` and `IEV` respectively.

This procedure can be used in nodes of any type.

Examples:

1.
```
ACT(SubwayDepart, 0); // The Subway departs now
```

2.
```
ACT(Controller, 7);   // Will be activated in 7 time units
```

3.
```
FOR(i=1; i<80; i++) ACT(Arrivals[i], EXPO(TMBarr[i]));
```

The components of the multiple node `Arrivals` are activated for future times taken from an exponential distribution with mean values depending on the node.

See examples 1 – 21 (GALATEA examples book)

6.2 ASSEMBLE assembles messages in a representative message

Syntax:

```
ASSEMBLE( | ELIM | FIRST | NEW |, |<integer value> | ALL|, |<logic
expression> | EQUFIELD(<field>) | ) <associated instruction>;
```

The `ASSEMBLE` instruction is used to join a set of messages that came to a node, maybe at different times, and to represent the set by only one message, named representant. The representant may be the first assembled or a new message. Represented messages may be deleted or maintained in a list for further disassembling. The associated instruction must contain a `SENDTO` instruction to dispose of the representant. The instruction is suited to simulate loading and transportation of things, using the representant as the transporter.

ASSEMBLE can be used in a node with an `EL` that has the ability to send messages (nodes of G, L, D, and general type with `EL`).

When this instruction is executed, the `EL` is scanned from the beginning. Each message (comparing message) is compared with those (compared messages) that follow it. The first comparing message is the first one in the `EL`. The compared message is named `OMESS`. The comparison consists in the evaluation of the `<logic expression>` that may include comparing message and `OMESS` fields, and any other type of variables and constants. If the `<logic expression>` is `TRUE` both, the comparing and compared messages, are candidates to be assembled. When all messages are compared, the second message in the `EL` is taken as the new comparing message, which is compared with those that follow it. The process is repeated so that all the possible comparisons are made. If at any point of the process a number of candidates reach the value expressed by `<integer value>`, the assemble is successful. Then, the assembled messages are taken out of the list and the `<associated instruction>` is executed. The whole process is then repeated to see if another successful assemble group of the required size is possible with the remaining messages. If the parameter `ALL` is used, there is not limit for the assembled set. If the `<logical expression>` is reduced to the constant `TRUE`, the messages are unconditionally assembled up to the specified number (or all the messages if `ALL` is used). If there is not successful assemble at all, nothing is done and the `<associated instruction>` is not executed. If the function `EQUFIELD(<field>)` is used instead of the `<logical condition>`, the messages, which have the same values on that field, are assembled. In all cases, it is assumed that all the messages in the `EL` have the fields that are used by the assembling conditions. Which message is the representant and the fate of the assembled messages depends on the first parameter:

- `ELIM` indicates that the representant is the first of the assembled group and the remaining of the group are eliminated.
- `FIRST` indicates that the representant is the first of the assembled group and the rest are put in an assembled list pointed by the representant. This allows the retrieval of them by a `DEASSEMBLE` instruction. If the representant has yet an assembled list from a previous assemble process, the new assembled messages are added to this list.
- `NEW` indicates that a new message is created as representant of the assembled group. The values of its fields are copied from the first of the group, but they can be changed in the `<associated instruction>`. This new message points to the assembled list so that it may be later retrieved through a DEASSEMBLE instruction.

The `<associated instruction>`, executed for each successful assemble, may have instructions to change the fields of the representant (assigning values to field variables) and must have a `SENDTO` instruction to dispose of the representant. Note that this `<associated instruction>` may change the values of variables in the `<logical condition>` and in the `<integer value>` so changing the assemble conditions for the next group or for a new execution of the `ASSEMBLE` instruction. Assembled messages cannot be changed. If there was some successful assemble the variable `SYNC` is put to `TRUE`, otherwise it is put to `FALSE`. See the instruction `SYNCHRONIZE` (6.26).

Examples:

1.
```
Dispatcher (G) {
  ASSEMBLE(FIRST,TRUE) SENDTO(Truk);
}
```

The `Dispatcher` assembles the order made up of all the messages that are on the `EL` list and sends it to the `Truck`.

2.
```
UniteParts (G){
  ASSEMBLE(ELIM, 3, EQUFIELD(typed)){
    status = finished;
    SENDTO(Paint);
  }
}
```

Groups of three messages with equal values in the field `typed` are sought for. When a group is found, the first message is marked as `finished` in its field `status` and sent to the `EL` of the node `Paint`. The other two messages of the group are deleted. The node `UniteParts` is of the common type.

3.
```
FormGroups (G){
  ASSEMBLE(FIRST, N,(age > 10) AND (O.age > 10)){
    leader = TRUE;
    SENDTO(Tour);
  }
  IF(NOT SYNC AND (N > 2)) N = N - 1;
}
```

The process attempts to form groups of size `N` (whose value was defined elsewhere) of messages with the value of the field `age` > 10. When a group is found, the first one is marked putting to `TRUE` its field `leader` and sent to the node `Tour`, keeping a point to the assembled messages. All the possible groups of this size are formed by the `ASSEMBLE` instruction. When the instruction is finished, as the node is of G type, the process is repeated if the `EL` is not empty starting again for the first message of the `EL`. When an attempt to form groups is not successful, the variable `SYNC` is put to `FALSE` by the `ASSEMBLE` instruction and the size of the sought group is reduced by one. The process is repeated until groups of size two are sought for.

See examples 9, 11, 12, 13 (GALATEA examples book)

6.3 COPYMESS makes copies of a message

Syntax:

```
COPYMESS(<integer value>) <associated instruction>;
```

This instruction produces a number of copies of the message that is being processed, equal to the value of `<integer value>` (that must be greater than zero) and for each copy executes the `<associated instruction>`. All fields of the original are copied.

The `<associated instruction>` must have a `SENDTO` instruction to dispose of the copy. Before it, instructions can be used to change the values of the fields. To do this the names of the `O.<fields>` must be used.

The original message must be processed through instructions outside the `<associated instruction>` before or after the `COPYMESS`. If it is not sent to a node, it will be lost.

The variable `ICOPY` takes the value of the generated copy. Its value is not recorded in the copied message. If the programmer wants to keep it, a field must be defined in the message for this purpose and the value of `ICOPY` must be passed to each copy in the `<associated instruction>`.

`COPYMESS` can be used only in G, I, D nodes and in the `RELEASE` instruction of an R node.

Examples:

1.
```
COPYMESS(nOffices){
  nc = ICOPY + 1;
  SENDTO(Office[ICOPY])
}
    SENDTO(Archive)
```

A number of copies equal to the actual value of `nOffices` are generated and sent to the nodes Office[0], Office[1], etc. The (`nc`) keeps the number of copy, with an indication of its `ICOPY` plus one. The original is sent to `Archive`.

See examples 9, 11 (GALATEA examples book)

6.4 CREATE creates a message

6.5 EXTR extracts a message from a list

Syntax:

```
EXTR(<message list>, <|message|FIRST|LAST|>)
```

It is used to extract a message from a list of messages. `<list name>` indicates the list. The second argument could be a `<message>`, a variable declared of `MESSAGE` type, `FIRST` or `LAST`. The message (or the first or last) is extracted and identified as `OMESS`, and the `<associate instruction>` is executed. The `<associate instruction>` must have a `SENDTO` instruction to send the `OMESS` message to a list.

This instruction can be used in any type of node and in the associated instruction of a `RELEASE`.

Example:

1.
```
Window (R) {
  STAY(1);
  IF(LL(Depot.EL)<quantity) reject=TRUE;
  ELSE reject=FALSE;
  RELEASE{
    IF(reject) SENDTO(Rejected);
    ELSE {
      for(i=0; i<quantity; i++)
        EXTR(Depot.EL, FIRST) SENDTO(Dispatcher);
      SENDTO(Dispatcher);
    }
  }
}
```

The messages (orders) arrive at the `Window` queue. At the `Window` they take a unit of time to be served. An message (order) is `Rejected` if the `quantity` requested is greater than what is in the `Depot`. If the order is not rejected, then the requested `quantity` is extracted from the depot (one by one), sending each `OMESS` (item) to the `Dispatcher`, who also receives the message (order).

6.6 FILE writes values in a text file

Syntax:

`FILE(<file name>, <format>, <list of expression>);`

It is used to write data in a text file. `FILE` instruction relieves the user from declare, assign, open, and close the files.

`<file name>` is a valid name of file. If the file exists, it is overwritten by the new data. If it does not, then it is created and opened at the beginning of the simulation and closed at the end of the simulation.

`<format>` is a String that includes specifiers (subsequences beginning with %) to indicates how to write the values of the expressions in the file (See **??**).

`<list of expression>` is a list of expressions to file.

If many FILE instructions write in the same file, the order of the data in the file is the order of the executions.

This instruction can be used in any type of node.

Example:

```
NETWORK
  Aa (A) {
    FOR(INT i=0; i<3; i++)
      FILE("arxm.dat", "%8.3f %2c %3d %e %.0f\n", i+azz, ch1, i, b[i], azz);
  }
INIT
  ACT(Aa, 0);
  FILE("arxm.dat", "i+azz    ch1  i   b[i]      azz\n");
DECL
  DOUBLE azz = 99;
  CHAR ch1 = 'J';
  DOUBLE b[3] = {19, 20, 21};
END
```

The file `arxm.dat` will contain:

```
i+azz    ch1  i   b[i]      azz
  99.000  J   0 1.900000e+01 99
 100.000  J   1 2.000000e+01 99
 101.000  J   2 2.100000e+01 99
```

See examples 5, 7 (GALATEA examples book)

6.7 GRAPH

Syntax:

`GRAPH(<title>, <list of variable>);`

It is used to display graphics during the simulation run. When executed for the first time, it constructs the axis, reference lines, numbers on the axis, displays the names given to the variables and scales and marks the initial values of the variables to be graphed. In the following executions it adds a new point to the curves of the variables being graphed and unites the points to the the already displayed curves.

`<title>` is a string of characters. It identifies the graphic and will appear as a title of it.

`<list of values>` is a list of items describing the variables to be depicted. The list consists of the following elements, separated by commas:

- `<numeric expression>`, that expresses the value to be graphed.
- `<description>`, a string that will appear in the graph to describe the data.

`GRAPH` may be used in any node. By default, the first value is used as independent variable.

Examples:

1.
```
TITLE
  Sinusoidal curve
NETWORK
  Fun (A) { IT = tf; y = SIN(PI * TIME); ACT(Gra, 0); }
  Gra (A) {
    GRAPH("Sinusoidal curve", TIME, "x", y, "sin(PI x)");
      }
INIT
  TSIM = 20; ACT(Fun, 0);
  tf = 0.0150625;
DECL DOUBLE tf, y;
END.
```

   A sinusoidal curve is displayed.

2.
```
TITLE
```

```
  Simple bank teller
NETWORK
  Entrance (I) { IT = EXPO(tba); ACT(Gra, 0); SENDTO(Teller); }
  Teller (R)   { STAY = GAUSS(mst, 2); RELEASE SENDTO(Exit); }
  Exit (E)     { ACT(Gra, 0); }
  Gra (A)      { GRAPH("Teller queue length", TIME, "time",
                       LL(Teller.EL), "Queue"); }
INIT
  TSIM = 480; ACT(Entrance, 0);
  INTI(tba, 3, "Mean time between arraives");
  INTI(mst, 3, "Mean service time");
DECL DOUBLE tba, mst;
END.
```

The program is a simulation of a simple bank teller and a graphic of the queue is made. Each time a client enters or leaves the `Teller`. the `Gra` node is activated and the length of the queue (`EL` of `Teller`) is put into the graph.

6.8 IF Specifies that a set of instructions to be executed or not

Syntax:

```
   IF <logic expression> <associated instruction 1>;
IF <logic expression> <associated instruction 1> ELSE <associated
instruction 2>;
```

Executes the `<associated instruction 1>` if the `<logic expression>` is truthy. If the condition is falsy, the `<associated instruction 1>` is not executed and if the `ELSE` part exists, the `<associated instruction 2>` is executed.

If the `<associated instruction 1>` has more than one instruction, the set of instructions must be enclosed in curly brackets. The same applies for `<associated instruction 2>`.

Examples:

1. `IF LL(Teller.EL)>20 NM20=NM20+1;`

The variable `NM20` is incremented by 1 if the length of the entry list of node `Teller` is greater than 20.

2.
```
IF(i==j) {
            j=10;
            IF(i<k){
                j++;
            }
        }
```

As two instructions are executed if i==j, both enclosed in curly brackets. The first is `j=10;` and the second, `IF(i<k)`, is another `IF` statement, which in turn has its own associated statement. Note that in the second `IF` the curly brackets are not needed, however it is not an error to use them.

3.
```
     IF((LL(Gb.EL)) > 15) {
            SENDTO(Exit);
        } ELSE IF(priority < 4) {
            SENDTO(Gb,FIRST);
        } ELSE {
            SENDTO(Gb);
        }
```

If the length of the entry list of the `Control` node is greater than 15 the message is sent to the `Exit` node. Otherwise, the message is sent to the `Control` node, placing it first in the entry list if the `priority` field is less than 4, but if it is not, it is located according to the established discipline of the entry list of `Control` node.

See examples 1, 2, 4, 7 – 13, 15, 18, 19 (GALATEA examples book)

6.9 IT schedules next activation of the node

Syntax:
```
IT(<real expression>);
```

This instruction is used to activate the node in which it appears after an interval time given by the current value of `<real expression>`. When executed, the value of `<real expression>` is assigned to the variable `IT` and an event is put in the future event list (`FEL`) with:

- The the node being executed.
- An index equal to the current value of `INO` (0 for single nodes 0, 1, 2, 3, . . . for successive executions of a multiple node).
- Time equal to the current value of `TIME + <real expression>`, that is to say `TIME + IT`.

When the node is multiple, the instruction may be executed many times. In each execution an event is entered in the `FEL`. The successive values of the index are 0, 1, 2, 3, . . . etc.

See instruction `NT` and procedure `ACT`.

The instruction can be used in any node, except of C type. Examples:

-
  ```
  Ent (I) {
    IT(UNIF(1.00, 1.03));
    SENDTO(Lathe[MIN]);
  }
  Lathe[4] (R) {
  ```

  Messages (parts) are generated each time the node `Ent` is activated. This happens at intervals taken at random from a uniform distribution between 1.00 and 1.03. The messages go to the less crowded of the four nodes `Lathe`.

-
  ```
  Signal (I) {
    IF(TIME < 1999) IT(1999 - TIME)
    ELSE WRITE('This is the last signal');
    SENDTO(NextNode);
  }
  ```

  If the node is activated before the `TIME` = 1999, then an activation is scheduled for the time 1999. if not, the message `This is the last signal` is written.

-
  ```
   Gate[3] (G){
  ```

```
      STATE{
        OPEN = NOT OPEN;
        IT(interval[INO]);
    }
    IF(OPEN) SENDTO(Deposit[INO]);
```

Incoming messages are retained each in its gate while the gate is closed and allowed to passed to `Deposit` when the gate is open. The gates are alternatively closed and open. The intervals of each state `OPEN` and `NOT OPEN` are equal for a gate but are different for the different gates. For the first gate they are `interval[0]`, for the second `interval[1]`, for the third `interval[2]`.

See examples 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 13, 14, 15, 17, 18 (GALATEA examples book)

6.10 PARAM allows interactive change of data

Syntax:
`PARAM(<variable>, <default value>, <description>);`
`PARAM(<gFunction>, <list of points>, <description>);`

This instruction allows you to define and set the values of the parameters at the beginning of simulation.

`<variable>` is the name of a simple variable.

`<default value>` is the default value given to the variable. The `<variable>` and `<default value>` data types must be compatible.

`<description>` is a string that contains a description of the variable, It is used in the model help message and in the graphic user interface of the model.

`<gFunction>` is the name of a GFUNCTION (See **??**).

`<list of points>` is a list of pairs of values (points). The first element of the pair is the value of the argument (it belongs to the function domain). The second is the corresponding value of the function.

Examples:

1. `INIT`

```
          PARAM(Alfa, 1.058, "Scattering angle");
                PARAM(FTB,(0.0,-0.2)(0.3,-0.1)(0.5,0.0)(0.8,0.1)(1.0,0.2), "FTB Pt
```

See examples 1, 5, 6, 7, 8, 10, 12, 13, 14, 15, 16, 18, 19, 20, 21 (GALATEA examples book).

6.11 REL

6.12 RELEASE

Syntax:

`RELEASE / ALL / <associated instruction>;`

This instruction is used to manage the messages that abandon the IL of a R type node, when the scheduled time is over. It is used when the default disposition of the message is not desired. The user may program a different procedure in the `<associated instruction>`. This may includes changes in the fields, conditional sending to different nodes, and a `NOTFREE` instruction to inhibit the freeing of the used resource.

It is executed only if the node is activated by an event. The execution is not repeated during the scanning of the network.

When the instruction is executed, a scanning of the `IL` starts, looking for a message with the exit time of the `IL` (that is recorded in the field `XT`). If such a message is not found, an error condition occurs. If it is found, its fields are passed to the corresponding field variables and the `<associated instruction>` is executed. This instruction must execute a `SENDTO` instruction to dispose of the message.

In the process (unless `NOTFREE` is used) a quantity of resource is freed that is equal to the value of the `USE` parameter of the message. So, the variable `<node>.FREE` is increased by `USE` and `<node>.USE` is decreased by `USE`. If the capacity of the resource is the constant `MAXREAL`, the resource is considered of infinite capacity and no change in those variables is made.

If the `ALL` parameter is used, the search for other messages with the same `XT` continues until all of them are processed.

This instruction can be used in R type nodes.

Example:

```
Physician (R){
```

```
STAY(TRIA(4, 6, 9));
RELEASE{
  Revised = TRUE;
  SENDTO(Treatment[Disease]);
}
```

Messages (patients) that leave the node `Physician` after completing its scheduled time of attention (given at the entrance by the instruction `STAY`) are tagged putting `TRUE` its field Revised. Then they are sent to one node of the multiple node `Treatment` according the value of their field `Disease`.

See examples 1-11 and 13 (GALATEA examples book).

6.13 SCAN

6.14 SENDTO sends messages in process to lists

Syntax:

`SENDTO(<destination list>, < | <pointer>, | A | B |> | FIRST | LAST |> );`

This instruction is used to send messages to one or more lists of messages. `<destination list>` is a list of names of nodes or lists. They may be of the form `<node>`, `<node>.EL` or `<node>.IL`.

Message in process is sent to all these destinations, if the list has more than one element, copies are sent at each destination.

If destination is a node, it is assumed that the message must be sent to the `EL` of the node.

The place in the destination list in which the message is put, depends on the second parameter:

- `LAST` the message will be appended at the end of the list.
- `FIRST` the message will be put at the beginning of the list.
- `BEFORE <pointer>` the message is put before the pointed message.
- `AFTER <pointer>` the message is put after the pointed message.

If the first parameter is omitted, `LAST` is assumed.

If this instruction is within an `<associated instruction>` of a GALATEA instruction, then the `OMESS` message is send, except for the `CREATE` in which MESS is send.

In the case the `SENDTO` is in an `<associated instruction>`, the sent message is:

- In `ASSEMBLE`: the representant of the assembled messages.
- In `COPYMESS`: each copy made of the original message.
- In `CREATE`: the created message.
- In `DEASSEMBLE`: each message that is taken off the assembled list.
- In `EXTR`: the extracted message.
- In `PREEMPTION`: the removed (preempted) message.
- In `REL`: the extracted from the IL.
- In `RELEASE`: the extracted from the IL.
- In `SCAN`: the actually examined message.
- In `SELECT`: each selected message.
- In `SYNCHRONIZE`: each synchronized message.

`SENDTO` can be used in type G, I, D, E, R, A, and general type nodes, and in the `<associated instruction>` of the GALATEA instructions indicated above, such as in the `RELEASE` instruction.

If the destination is a multiple node, the user have three alternatives to send a message to its `EL`s:

1. To indicate the desired index in the node specified in the `SENDTO`.
2. To use the reserved word `MIN` as index. In this case, the message is sent to the first node for which the sum: Length of the `EL` plus Quantity of used resource is the minimum. That means the less engaged node of the array.

3. To use the reserved word `FREE` as index. In this case, the message is sent to the `EL` of the first node that has the `EL` empty. If all the `EL`s have messages, the message is not sent. This option only may be used in the code of a type G node. Not sent messages remain in the `EL` of the node.

Examples:

1. `Mail (D) :: SENDTO(Director, Secretary, File);`

   Copies of the message received at Mail are sent to the nodes Director, Secretary, and File.

2.
```
Discrim (G){
      IF(Class == Friend) SENDTO(FIRST, Queue);
      ELSE SENDTO(LAST, Queue);
    }
```

3.
```
    Parkman (G) {SENDTO(Parkinglot[MIN]);}
Parkinglot (R) {
  STAY = GAMMA(Tpark, DevTpark);
  RELEASE SENDTO(Exit);
    }
```

4.
```
     Ii (I) { IT(1); Pr = UNIFI(1, 4); SENDTO(OrdQueue); }

     OrdQueue (G){
       IF(LineGate == Closed) {
         SCAN(Line.EL, poin){
           IF(Pr > OMESS.pr) STOPSCAN;
           IF(Poin != NIL) SENDTO(Line, BEFORE, Poin);
           ELSE SENDTO(Line, LAST);
          }
       ELSE SENDTO(Line);
        }

     Line (G){
       STATE{
         IF{LineGate == Closed} LineGate = open;
```

```
      ELSE LineGate = Closed;
      IT(8);
    }
  IF(LineGate == Open) SENDTO(Process);

  INIT
    TSIM = 100; ACT(Ii, 0); LineGate = Open; ACT(Line, 8);

  DECL TYPE Opcl = (closed, open);
  VAR LineGate: Opcl; Poin: POINTER;
  MESSAGES Ii(INT Pr);

  END.
```

Messages, with priorities from 1 to 4 in the field `pr`, arrive at `OrdQueue` and they are passed to `Line` if the variable `lineGate` has the value `open`. In this case, they continue to `Process`. `lineGate` is `open` and `closed` each 8 time units. If `lineGate` is `closed`, then when a message reaches `OrdQueue`, a scan process of the `EL` of `Line` starts. If a message in the `EL` is found with priority less than the incoming message, then this is put before of that in the `EL`. So, the `EL` is always ordered in decreasing order of priority `pr`. In this system the entities that arrive when the gate is open are allowed to pass, when they have to wait, they are sorted according to their priority. This type of sorting may be more easily done using a type L node.

See examples 1 – 13, 18 (GALATEA examples book).

6.15 STATE

6.16 STAY

Syntax:

```
STAY(<real expression>;)
```

It is used to assign value to the time that a processed message will remains in the IL of a R type node.

It is only used in a R type node.

When executed a message enters the `IL`, an event is scheduled in the future to extract this message. More exactly, an event element is introduced in the `FEL` that refers to the R type node, to be executed at a time `TIME` + `STAY` (with the assigned value for the `STAY`). The index of the event is taken from the variable `INO` and the event parameter will be the actual value of the variable `IEV`. If the node contains a `PREEMPTION` instruction with a `REMA` parameter, the message must have a `REMA` field. The system checks the value of this field. If its value is not equal to zero, the time of execution of the event will be `TIME` + `REMA`. See `PREEMPTION` instruction (6.17).

See examples 1, 2, 3, 4, 5, 6, 7, 8 (with PREEMPTION and REMA), 9, 11, 12, 13, 17, 18 (GLIDER examples book).

6.17 SYNCHRONIZE

6.18 TSIM

Syntax:

```
TSIM(<real expression>);
```

It is used to set the simulation duration. The GALATEA system variable `TSIM` is set to the value of `<real variable>` and the end of the simulation is scheduled for a future time equal to `TSIM`. The end of the simulation can also be done using the `ENDSIMUL` instruction. If many `ENDSIMUL` instructions are present, the first one executed will end the simulation. If many `TSIM` instructions are executed, the last executed defines the simulation time.

This instruction can be used in nodes of any type and in the `INIT` section.

6.19 USE

6.20 WRITE Print Formatted String to Console

Syntax

```
WRITE(format, arguments);
```

Writes the string `format` to the console. If `format` includes specifiers (subsequences beginning with %), the additional arguments following `format` are formatted and inserted in the resulting string replacing their respective specifiers.

Format Specifiers

Let's look at the available format specifiers available for `WRITE`:

- %c character
- %d decimal (integer) number (base 10)
- %e exponential floating-point number
- %f floating-point number
- %i integer (base 10)
- %o octal number (base 8)
- %s String
- %u unsigned decimal (integer) number
- %x number in hexadecimal (base 16)
- %t formats date/time

Escape Characters:

Following are the escape characters available in WRITE

- \b backspace
- \f next line first character starts to the right of current line last character
- \n newline
- \r carriage return
- \t tab
- \\ backslash

Format Specifiers Full Syntax

Let's look at the full syntax of format specifiers with the extended set:

`%<flags><width><.precision>specifier`

Examples

1.
```
     Test (A) {
       WRITE("Preceding with blanks: %10d \n", 1977);
       WRITE("Preceding with zeros: %010d \n", 1977);
       WRITE("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100,
       WRITE("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
       WRITE("%s \n", "A string");
 }
```

Chapter 7

Procedures

7.1 BEGINSCAN

7.2 CLRSTAT

Syntax:
`CLRSTAT;`
It is used to reset to zero the variables in which the statistics are accumulated. Thus the statistics are counted since the last time in which a procedure of this type is executed. That time is recorded in the standard statistics.

This procedure can be used in nodes of any type.

Examples:

```
   PassTrans (A){
 IF(ABS(qma - MEDL(Tell.EL)) / (0.5*(qma + MEDL(Tell.EL))) < 0.03){
   stable = TRUE;
   CLRSTAT;
 } ELSE {
   qma = MEDL(Tell.EL); CLRSTAT;
   IT = 1000;
     }
   }
```

The queue `Tell.EL` is examined. The previous mean length `qma` is compared with the mean length in the last 1000 time units. This is given by the GALATEA function `MEDL`. If the difference, compared with the mean of the two values, is less than 3%, the queue is considered `stable`, a new count of statistics begin. Otherwise, the last value of the mean is saved in `qma`, and a new statistical count is initialized for the next 1000 time units. Statistics at the end of the run are counted since the beginning of the stable situation. This procedure tries to avoid the influence on statistics of the transient produced when starting with a void queue.

7.3 DEACT

7.4 ENDSIMUL

7.5 METHOD

Syntax:

```
METHOD(| <RKF> | <EULER> | <RK4> |);
METHOD(<node name>, | <RKF> | <EULER> | <RK4> |);
```

It is used to set the integration method of the differential equations in type C nodes. If `RKF` is used, the method will be a Runge Kutta method of fourth order with a fifth order calculation to estimate the error (Runge Kutta Fehlberg). The integration path is changed according to the error. If `RK4` is used, the method will be a Runge Kutta of fourth order. If `EULER` is used, the method will be the simple Euler method of the first order. `RKF` is the more exact and slower. `EULER` is the less exact and faster.

`EABS` and `EREL` are used with RKF method of integration to change the integration step when the error in one step exceeds, respectively in absolute or relative value, the values of these variables. Default options are `EABS` = 0.00001 `EREL` = 0.0001.

`<node name>` must be the name of a type C node. If `node name> is not specified, this instrucción may be used only in C type nodes.`

`Example:`

```
Growth (C) {
  METHOD(RK4);
  size'     = kGrowth * (1 - size / maxSize) * size;
  quantity' = maxQuant - kIncr * quantity;
  biomass   = spGr * size * quantity;
}
```

```
HeatEffect (C) {
  METHOD(EULER);
  kGrowth' = kGrowthMax - ctg(temp) * kGrowth;
  kIncr'   = kIncrMax - cti(temp) * kIncr;
}
```

The Growth node uses the RK4 method and the HeatEffect node uses the EULER method. This may be so because it is supposed that kGrowth and kIncr growth very slowly with temp so that a less exact but more fast integration method may be used.

7.6 ORDER

7.7 PAUSE

Syntax:

PAUSE;

It is used to stop the execution of a simulation run. The user can restarts the simulation by pressing the continue button.

This procedure can be used in nodes of any type.

7.8 RETARD

7.9 STOPSCAN

7.10 HISTOGRAM

Syntax:

HISTOGRAM(<title>, <nBins>, <expression>, <description>);

It is used to count a value for a frequency table; <title>. The actual value of the expression is classified to define to which of the <nBins> frequency class of the table it belongs and one is added to the count of this class. The frequency classes and a histogram are shown during the simulation.

This procedure can be used in nodes of any type.

Example

```
   QueueStat (A){
     TAB("Length of the queue", 10, LL(Couter.EL);, "People");
   }
```

Messages (people) queue at the node Counter (not shown). The node QueueStat is actived when the queue changes. The length of the queue LL(Couter.EL) is registered at the frequency table "Length of the queue". This table has 10 classes (intervals).

7.11 TRACE

Syntax:
   TRACE;
It is used to put the trace mode that displays a detailed account of the events process, messages movements, FEL evolution, etc., useful for debugging.

This procedure can be used in nodes of any type and in INIT section.

Example:

```
    IF{Shop.EL > 10} TRACE; ELSE IF(Shop <= 5) UNTRACE;
```

If the queue in Shop is greater than 10 the trace starts, when the queue decreases and becomes 5 or less the trace is stopped.

7.12 UNTRACE

Syntax:
   UNTRACE;
It is used to stop the trace mode that displays a detailed account of the events process, messages movements, FEL evolution, etc., useful for debugging.

This procedure can be used in nodes of any type.

It must not be executed if the system is displaying a graphic.

Example:

```
    IF(TIME > 2000) UNTRACE;
```

Tracing mode is suppressed after the time 2000.

Chapter 8

Functions

8.1 Message list function

8.1.1 LL

```
Syntax:
   LL(<list name>);
Returns the value (INT) of the length of the list (EL or IL) indicated
in <list name>.  Example:
```

```
    IF(LL(Clerk1) <= 2 * LL(Clerk2)) SENDTO(Clerk1);
    ELSE SENDTO(Clerk2);
```

8.1.2 MAXL

```
Syntax:
   MAXL(<list name>);
Returns the value (INT) of the maximum value attained for the list
(EL or IL) indicated in <list name>.
   Example:
```

```
WRITE("The maximum in the queue was %d\n’, MAXL(Dec.EL));
```

8.1.3 MINL

```
Syntax:
   MINL(<list name>);
```
Returns the value (INT) of the minimum value attained for the list (EL or IL) indicated in <list name>. Example:

```
MaxiFreeSpa[i] = ParkSpa[i] - MIN(Parking[i].IL);
```

8.1.4 MEDL

Syntax:
```
   MEDL(<list name>);
```
Returns the value (DOUBLE) of the mean of the length as a function of time for the list (EL or IL) indicated in <list name>. Example:

```
    IF(MEDL(Mac[1].EL) > 1.5 * MEDL(Mac[2].EL))
        CAP(Mac[1], 2 * Mac[1].CAP);
```

8.1.5 DMEDL

Syntax:
```
   DMEDL(<list name>);
```
Returns the value (DOUBLE) of the standard deviation of the mean of the length as a function of time for the list (EL or IL) indicated in <list name>. Example:

```
    WRITE("Coefficient of variation for queue %5.2f\n",
           DMEDL(Office.EL) / MEDL(Office.EL));
```

8.1.6 MSTL

Syntax:
```
   MSTL(<list name>);
```
Returns the value (DOUBLE) of the mean of the waiting time of the messages that left the list (EL or IL) indicated in <list name>. Example:

```
    IF(MSTL(EL_Park[1]) < MSTL(Park_[2])) SENDTO(Park[1]);
    ELSE SENDTO(Park[2]);
```

8.1.7 DMSTL

Syntax:
```
   DMSTL(<list name>);
```
Returns the value (DOUBLE) of the standard deviation of the mean of the waiting time of the messages that left the list (EL or IL) indicated in <list name>. Example:

```
    IF(DMSTL(Door_A.EL) > DMSTL(Door_B.EL) door_name = "A"
    ELSE door_name = "B";
    WRITE("Queue in door %s had greater fluctuation", door_name);
```

8.1.8 TFREE

Syntax:
```
   TFREE(<list name>);
```
Returns the value (DOUBLE) of total time in which the list (EL or IL) indicated in <list name> was void. Example:

```
Access (I) {
        IT = UNIF(5, 7);
        prob_A = (1+TFREE(A.EL))/((1+TFREE(A.EL))*(1+TFREE(B.EL)));
        IF(BER(prob_A)) SENDTO(A);
        ELSE SENDTO(B);
    }
```

8.1.9 ENTR

Syntax:
```
   ENTR(<list name>);
```
Returns the value (INT) of the number of entries in the list (EL or IL) indicated in <list name>. Example:

```
    IF(ENTR(Theater.EL) > 120) BLOCK(Th_Arrival);
```

8.2 Node function

If it is a simple node, <node> is the identifier of the node. On the other hand, if the node is multiple, <node> must contain the node identifier and its index between square brackets.

8.2.1 CAP

```
Syntax
   CAP(<node>)
Returns (DOUBLE) the total quantity of resource of the R <node>.
```

8.2.2 DT

```
Syntax
   DT(<node>)
Returns (DOUBLE) the integration step that will be used in the indicated
C type node.  The RKF method may change it during the integration
process.
```

8.2.3 FREE

```
Syntax:
   FREE(<node>)
Returns (DOUBLE) the actual quantity of free resource of the R node.
Example:

    Control(G) {
      IF((FREE(Pier[ShiTyp]) > 0) AND (FREE(Channel) > 0))
        SENDTO(Channel);
      }

If there is some free capacity in Pier[ShiTyp] and in Channel the
message (ship) is sent to Channel.  Both nodes, Pier and Channel
must be type R. Pier is a multiple node and Channel is a single node.
```

8.2.4 USE

```
Syntax:
   USE(<node>)
Returns (DOUBLE) the actual quantity of used resource of the R node.
Example:

cost = cost + 2 * USAGE(Machine);

To the cost is added two times the USAGE of the Machine.
```

8.3 MAX

8.4 MIN

8.5 Random variable generators

8.5.1 BER

```
Syntax:
   BER(<real expression>, / <stream> /);
   Returns the BOOLEAN value TRUE with a probability equal to the
value of <real expression> and FALSE with a probability of (1 - <real
expression>).  The probabilities are computed with random numbers
taken from the stream indicated by <stream>.  This is an integer
from 0 to 9.  If omitted, 0 is assumed.  Example:

    IF(BER(0.6)) SENDTO(LargeDep); ELSE SENDTO(SmallDep);

At random 60% of the times the messages are sent to LargeDep.  40%
to SmallDep.
```

8.5.2 BETA

8.5.3 BIN

8.5.4 ERLG

8.5.5 EXPO random value from an Exponential distribution

```
Syntax:
   EXPO(<real expression> / <,stream> /);
Returns a DOUBLE value of a random variable whose probability function
is an exponential function with mean equal to <real expression>,
computed with random numbers taken from the stream indicated by <stream>.
This is an integer from 0 to 9.  If omitted, 9 is assumed.
   See examples 1, 3, 4, 5, 7, 13 (GALATEA examples book)
```

8.5.6 GAMMA

8.5.7 GAUSS

Syntax:
```
   GAUSS(<real expression>, <real expression> / <,stream> /);
```
Returns a DOUBLE value of a random variable whose probability function is a normal function in which only the positive values are considered. The mean is equal to the first <real expression> and the standard deviation equal to the second <real expression>, computed with random numbers taken from the stream indicated by <stream>. This is an integer from 0 to 9. If omitted, 0 is assumed.

8.5.8 LOGNORM

8.5.9 NORM

Syntax:
```
   NORM(<real expression>, <real expression> / <,stream> /);
```
Returns a DOUBLE value of a random variable whose probability function is a normal function with mean equal to the first <real expression> and deviation equal to the second <real expression>, computed with random numbers taken from the stream indicated by <stream>. This is an integer from 0 to 9. If omitted, 0 is assumed. The algorithm may produce negative values.

8.5.10 POISSON

Syntax:
```
   POISSON(<real expression> / <,stream> /);
```
Returns an INT value of a random variable whose probability function is a Poisson function with mean equal to <real expression>, computed with random numbers taken from the stream indicated by <stream>. This is an integer from 0 to 9. If omitted, 0 is assumed.

8.5.11 TRIA

Syntax:

```
   TRIA(<real expression>, <real expression>, <real expression> /
<,stream> /);
```
Returns a DOUBLE value of a random variable whose probability function is a triangular function with minimum equal to the first <real expression>, mode equal to the second <real expression>, and maximum equal to the third <real expression>, computed with random numbers taken from the stream indicated by <stream>. This is an integer from 0 to 9. If omitted, 0 is assumed.

8.5.12 UNIF

Syntax:
```
   UNIF(<real expression>, <real expression> / <,stream> /);
```
Returns a DOUBLE value of a random variable whose probability function is a continuous uniform function with minimum equal to the first <real expression> and maximum equal to the second <real expression>, computed with random numbers taken from the stream indicated by <stream>. This is an integer from 0 to 9. If omitted, 0 is assumed.

8.5.13 UNIFI

Syntax:
```
   UNIFI(<int expression>, <int expression> / <,stream> /);
```
Returns an INT value of a random variable whose probability function is a discrete uniform function with minimum equal to the first <int expression> and maximum equal to the second <int expression>, computed with random numbers taken from the stream indicated by <stream>. This is an integer from 0 to 9. If omitted 0, is assumed.

8.5.14 WEIBULL

Syntax:
```
   WEIBULL(<real expression>, <real expression> / <,stream> /);
```
Returns a value of a random variable whose probability function is a Weibull function with first parameter equal to the first <real expression> and second parameter equal to the second <real expression>, computed with random numbers taken from the stream indicated by <stream>. This is an integer from 0 to 9. If omitted, 0 is assumed.

Chapter 9

Reserved Words

The GALATEA language defines a set of identifiers as reserved words. They cannot be used as identifier by the user.

9.1 Message variables

- GT real; generation time of the message being processed.
- O_GT real; generation time of the message being processed.
- ET real; time of entry of the message to the actual list.
- O_ET real; time of entry of the alternative message to the actual list.
- XT real; time of exit of the message from the actual list.
- O_XT real; time of exit of the message from the actual list.
- USE real; quantity of resource to be used by the message.
- O_USE real; quantity of resource to be used by the alternative message.
- NUMBER integer; generation number, in its origin node, of the message being processed.
- NODE string; name of the origin node of the message being processed.

- O_NUMBER integer; generation number, in its origin node, of the alternative message being processed.
- O_NODE string; name of the origin node of the alternative message being processed.
- REMA real; its value is the remaining time of a preempted message; although it must be declared by the user in a MESSAGES declaration, it is managed by the instruction PREEMPTION.

9.2 Event variables

Indexed node variable Control and state variables Node dependent variables Variables depending on user's variables Variables that may be initialized by the user Classes of declarations GLIDER or Pascal predefined types Pascal separators GLIDER types of functions (GFUNCTIONS) GLIDER separators Operators GLIDER instructions and procedures Pascal procedures GLIDER functions Pascal functions Colors Pascal constants Other reserved words Types of Node Constants

Chapter 10

Variables and Atributes

10.1 Model Variables

10.1.1 TIME (`DOUBLE`): its value is the actual simulation time

10.1.2 TSIM (`DOUBLE`): its value is the total simulation time

10.2 Node Atributes

The nodes of a GALATEA model have the following attributes:

- IL (Message list): designs the internal list of the node.
- EL (Message list): designs the entry list of the node.
- INO (INT); index of the node being processed.

In the following <node name> includes the index between [], if the node is multiple.

10.3 Message Atributes

- GT (DOUBLE): generation time of the message being processed.
- O.GT (DOUBLE): generation time of the message being processed.

- `ET (DOUBLE): time of entry of the message to the actual list.`
- `O.ET (DOUBLE): time of entry of the alternative message to the actual list.`
- `XT (DOUBLE): time of exit of the message from the actual list.`
- `O.XT (DOUBLE): time of exit of the message from the actual list.`
- `USE (DOUBLE): quantity of resource to be used by the message.`
- `O.USE (DOUBLE): quantity of resource to be used by the alternative message.`
- `NUMBER (INT): generation number, in its origin node, of the message being processed.`
- `O.NUMBER (INT): generation number, in its origin node, of the alternative message being processed.`
- `NODE (STRING): name of the origin node of the message being processed.`
- `O.NODE string; name of the origin node of the alternative message being processed.`
- `REMA (DOUBLE): its value is the remaining time of a preempted message; although it must be declared by the user in a MESSAGES declaration, it is managed by the instruction PREEMPTION.`