

# Validation of Statecharts Based on Programmed Execution

**João W.L. Cangussu**

e-mail: cangussu@dct.ufms.br

Department of Computer and Statistics - UFMS  
PO Box 649 - 79070-900 Campo Grande - MS / Brazil

**Rosângela D. Penteado**

e-mail: rosangel@icmsc.sc.usp.br

Department of Computer - UFSCar, IFSC  
PO Box 676 - 13565-905 São Carlos - SP / Brazil

**Paulo C. Masiero**

e-mail: masiero@icmsc.sc.usp.br

Department of Computer Science and Statistics - ICMSC-USP  
PO Box 668 - 13560-970 São Carlos - SP / Brazil

**José C. Maldonado**

e-mail: jcmaldon@icmsc.sc.usp.br

Department of Computer Science and Statistics - ICMSC-USP  
PO Box 668 - 13560-970 São Carlos - SP / Brazil

## Abstract:

Statechart is a graphical technique for specification of reactive systems. Validation of a statechart model is very important due to its potential complexity and can be supported by different types of simulations: interactive, batch, programmed, exhaustive and prototyping. In this paper we present a language named Execution Control Language (ECL) to support the simulation of models based on randomic generation of events and controlled execution. An example is shown and results obtained using programmed execution are analyzed and used to improve the first version of the statechart model.

## Keywords:

Simulation, Statecharts, Reactive Systems, Programmed Execution

Journal of Computing and Information

*Special Issue:* Proceedings of Seventh International Conference of Computing and Information (ICCI'95), Trent University, Peterborough, Ontario, Canada, July 5-8, 1995, pp. 870-885.

©1995, Journal of Computing and Information (JCI)

## 1- Introduction

Reactive systems are defined as systems that interact with the environment, receiving and issuing stimuli. Such systems must produce correct results within restrict time intervals and when this does not happen, a fault is characterized [1]. Reactive systems include communication networks, automobiles, telephones, digital watches and chips. While in their counterpart, the Transformational Systems, attention is focused on the initial and final states, in reactive systems all internal states are important, due to their continuous interaction with the environment [2].

Finite State Machines, Petri Nets and Statecharts are graphical techniques for specifying the dynamic behavior of Reactive systems. It has been shown that statecharts are very useful in the specification of such systems, which fostered the development of several environments based on statecharts, such as Statemate [3] and Argonaute [4]. Statechart is a visual formalism proposed by Harel [5] for the specification of complex reactive systems.

Several reports have been published on the use of statecharts for modelling reactive systems, e.g., avionics systems [6,7], computer operating systems [8] and man-machine interface management systems [9]. They provide an extension to conventional state-transition diagrams based on finite state machines whose outstanding features are hierarchy of states, the ability to specify parallelism and a communication mechanism via broadcasting. They also feature a rich set of special notations for augmenting the notational power of state-transition diagrams, thus allowing the specification of complex problems in very concise diagrams. This descriptive power makes it difficult to validate a statechart model against its intended behavior. In practice, system models specified through formalisms such statecharts may be validated in many ways; possible approaches are: batch, interactive and programmed execution, prototype execution and code generation [6].

Most of the environments for reactive systems specification include tools for the interactive execution of models described in graphical or textual form. Although useful for verification and validation of models, interactive execution, where the users interact with the system directly, providing input events and modifying variables and conditions, enables system execution in a free way, according to users requirements. It is also important to observe system execution under random conditions, rather than only with previously designed scenarios. This can be achieved using a simulation control program that generates events randomly, sets and changes variable values and captures statistical information about the execution [3]. Programmed execution complements other

simulation techniques, namely batch, interactive and exhaustive simulation, as it can produce results that are very difficult to obtain with these latter techniques [6].

The objective of this paper is to show a detailed example of statechart validation based on the Programmed Execution simulation technique within StatSim [10], a statechart environment developed in our research group. An Execution Control Language (ECL) was designed and an execution module integrated to StatSim allow this type of simulation. We also want to emphasize some statistical features embodied into ECL which were inspired by the simulation language GPSS (General Purpose Simulation System) [11,12] thus making StatSim suitable for handling "statistical" problems. The Statemate environment also contains a module to control the simulation of statechart models similar to the one presented here [3].

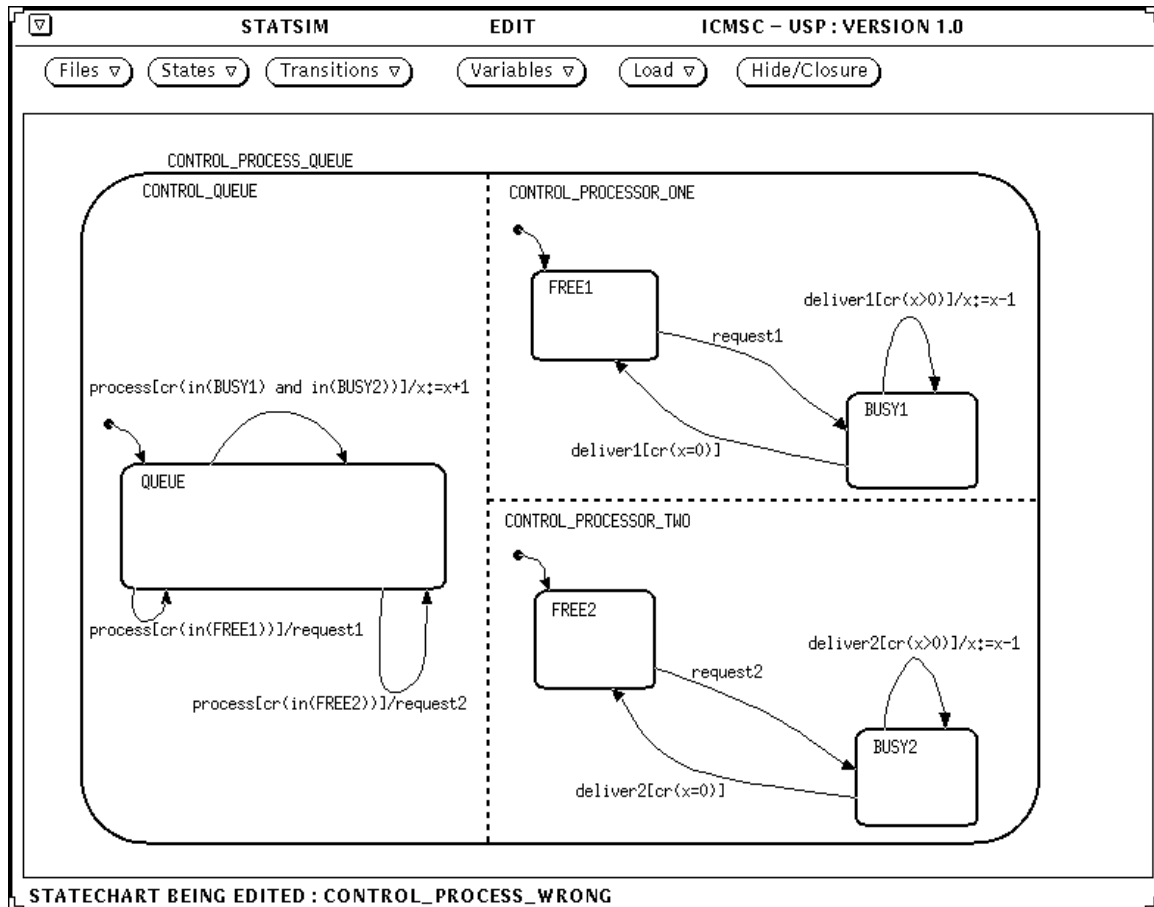
This paper is organized as follows: Section 2 presents a brief introduction to statecharts and Section 3 gives an overview of the StatSim environment. In Section 4 we present the main features of ECL and how to use it within StatSim. Section 5 presents an example comprising a statechart model, a program in ECL that simulates the statechart and an analysis of the reports produced. Concluding remarks are presented in Section 6.

## 2. An Overview of Statecharts

As with conventional state transition diagrams, states constitute the basic components of statecharts. However, in statecharts, states may be embedded into superstates thus creating hierarchies of states. Superstates may be of two types: AND-States or XOR-States. The former capture the notion of independence - image and sound may be on at the same time on a TV set. The latter correspond to refinement of states, e.g. a digital watch either displays time or date or acts as a stopwatch. The components of an AND-State are called orthogonal components and have the distinctive feature that a system at an AND-State it is also in all orthogonal components of such state. Conversely, if the system is at an XOR-State, it must be in only one of its substates.

The statechart of Figure 1 is intended to model the use of two processors by processes requesting them. Requests for processing are controlled by a queue. When a request for processing arrives, it is attended immediately if one of the processors is free, otherwise it is enqueued. The statechart of Figure 1 contains an AND-state **CONTROL\_PROCESS\_QUEUE** and three XOR-states: **CONTROL\_QUEUE**, **CONTROL\_PROCESSOR\_ONE** and **CONTROL\_PROCESSOR\_TWO**; variable *x* indicates the queue's size.

States interact through transitions composed by three basic parts: an event expression, a condition expression and action statements. These components appear as labels attached to the transitions according to the syntax: event-expression[condition]/{action}. The event expression may be null if there is a condition, and the action specification is also optional. Conditions are formed by a combination of logical and relational operators involving variables and special conditions, e.g., being in a state. Event-expressions are formed from a basic set of primitive events which can be combined using logical disjunctions and conjunctions.



**Figure 1 - Statechart to Control a Queue of Process**

The transition labeled **deliver1[cr(x=0)]** is fired at a certain time step only if the state **BUSY1** is active at that time, the event **deliver1** occurs and the current value of **x** is zero, i.e., its value during microsteps [14]. Execution of an action may generate other

events which are broadcast to the orthogonal components, possibly firing new transitions. Assignments of arithmetic and logical expressions to variables are also allowed as legal action statements. In Figure 1, the transition labeled **process[cr(in(FREE1))]/request1** is fired when **QUEUE** is active, **process** occurs and state **FREE1** is currently active. As result of firing this transition the event **request1** is generated and broadcast to the orthogonal components of **CONTROL\_PROCESS\_QUEUE** which may fire the transition from **FREE1** to **BUSY1**, depending whether or not **FREE1** was active at the beginning of that time step.

Execution of a statechart is based on a sequence of time steps where, at each step, events generated by the environment, added to the ones generated internally as a consequence of action execution, cause a set of transitions to be fired. The intended semantics is that firing a transition takes no time, while being in a state takes a certain time.

Amongst the various extensions added to conventional finite state machines, statecharts allow the specification of state history, several special events like entering or exiting an state (en, ex) or changes in the value of a control expression (tr,fs) or variable (ch) as well as the special conditions current (cr) and not yet (ny) to test the current state of a model and the occurrence of events within a time step.

### 3. The StatSim Environment

The StatSim environment comprises a graphical editor for creating statecharts and a set of tools for statechart simulation. The graphical editor allows insertion and edition of states and transitions as well as the specification of variables. We follow the syntax defined by Harel [5,14] and always try to enforce it when new elements are inserted in the model; global verifications can be made on demand. An internal event can be hidden as defined in [15], so that it cannot be seen from the environment.

States and transitions may have associated activities and actions, respectively. Activities are subject to three types of activation: **DO** — when they execute during the whole duration of the states; **ENTRY** — the activation occurs when the state is entered; and **EXIT** — the activation occurs when the state is left. Actions are activated instantaneously when their associated transitions are fired.

The simulation tools of StatSim allow several types of simulations: interactive, batch, programmed and exhaustive. In the interactive simulation the user specifies at each step the events that should be triggered thus creating an execution scenario; active states

are painted on the screen and a system animation is created as the scenario develops. All stepwise changes of system configuration are logged in a file which can be examined either on line or printed. Several options support the simulation: the user can query or change the variables values, clean history, query states and transitions, etc.

StatSim also supports exhaustive simulation which is based on a reachability tree where all configurations are computed [16]. Several algorithms are available to analyze statecharts in order to detect deadlocks, configuration reachability, transition usage, valid sequence of events and reinitiability. Storage and time restrictions may cause difficulties to analyze some large or complex models and analysis of partial statecharts is a possible solution which is encouraged by their hierarchical nature.

Programmed Execution with random generation of events [13,17] is discussed in detail in the remainder of this paper. Batch execution is a very simple form of programmed execution in our environment: the user devises a scenario through a sequence of events and StatSim animates it or displays the simulation step by step under the user control.

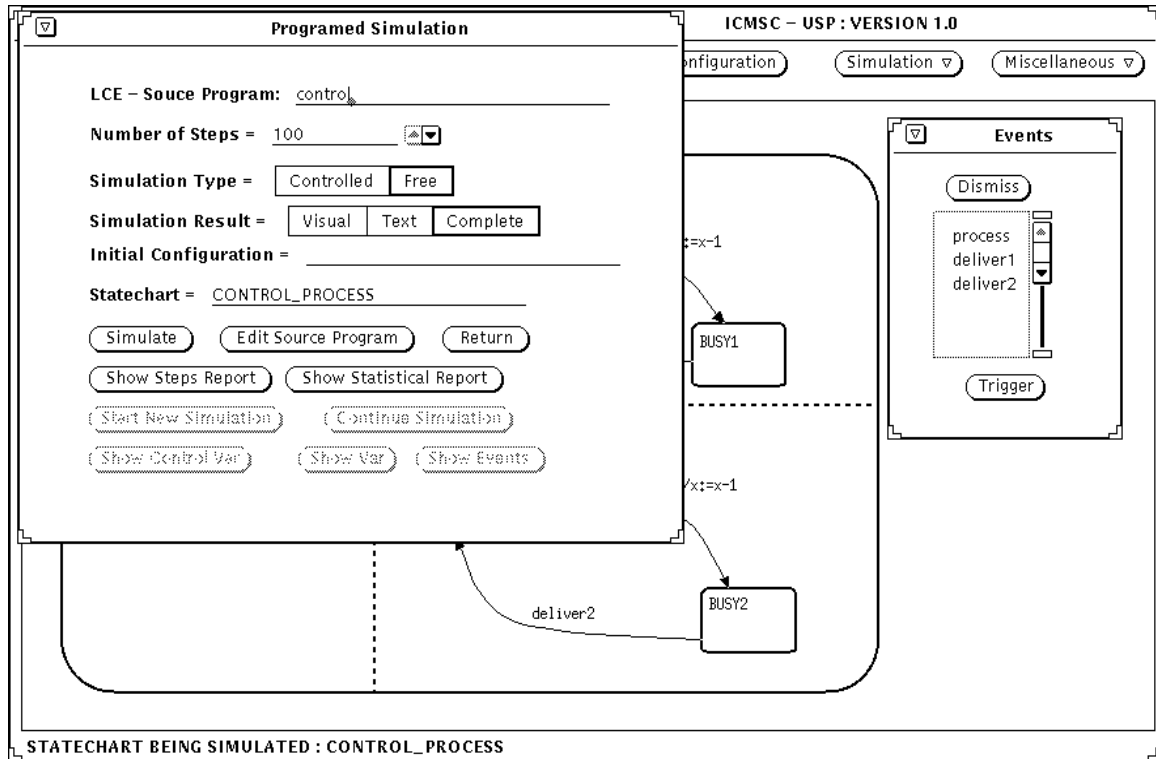
## **4. Programmed Execution**

Programmed execution is the simulation of a model conducted according to a previously devised script specified through a control program and with occurrences of events specified through probabilistic distributions that model their behavior in the real world. In StatSim the underlying model is a statechart and the control language is ECL. ECL in fact, is independent of the underlying model.

ECL is composed of three parts. In the first part global parameters of the simulation are defined, as are the number of simulation steps, the initial configuration, whether it should be performed stepwise (controlled by the user) or free (the simulation evolves automatically without user control). The user can specify these parameters in a ECL source program or fill in a form provided by StatSim. Default values are available for all global parameters. Figure 2 shows the interface for the example of Figure 1.

The second part of ECL allows declarations of control variables and the specification of events distribution. Control variables are of type integer and are used within the simulation program to record certain aspects of the simulation or to make decisions that affect the simulation flow. One should not confuse them with the statechart variables that are defined in the statechart and can also be handled inside any ECL program.

Ocurrences of external events are specified through the use of probabilistic distributions: normal, exponential and uniform. They can also be specified to occur at fixed intervals or to use a linear distribution. It is also possible simply to create a sequential file containing the desired event sequence. New events, not belonging to the statechart being simulated, cannot be created.



**Figure 2 -Initial Interface of MEP**

The third part of ECL specifies the simulation procedure (or script), basically defining certain desired states on certain specific steps and what to do when the simulation reaches those states or steps. A summary of the available commands in ECL is provided in Table 1. The user is able to program interruptions in the automatic execution of the statechart and to intervene using the facilities provided by the interactive execution.

Several commands were adapted to our needs from GPSS [11]. It should be noted that while GPSS is a language that builds a system' model by itself, this is not true for ECL, as our underlying model is a statechart and the language allows the specification of a script to drive and to collect data about the statechart simulation. A model is described in

GPSS as a block diagram in which transactions (temporary entities) flow from a creation point to an endpoint. This is, in a certain way, similar to the programmed execution.

**Table 1 - Summary of ECL Statements**

Statement	Meaning
AT STEP s1, ..., sn	Specify the statements that will be executed at steps s1, ..., sn
ALL STEPS	Specify the statements that will be executed at all steps
FOR STEP	Specify the statements that will be executed at the steps determined by the FOR statement
IF	Conditional statement
FOR	Loop statement
WHILE	Loop statement
SHOW_VAR	Show the statechart' s variables
SHOW_CONTROL_VAR	Show the control' s variables
SHOW_EVENTS	Show the events triggered at the current step
INTERACTIVE	Return to the interactive simulation
EVENTS(parameters)	Trigger the events passed as parameters
AVAIL(parameters)	Make available the events passed as parameters
UNAVAIL(parameters)	Make unavailable the events passed as parameter
MESSAGE	Print messages/values.
SET_CONFIGURATION	The specified configuration is set as the new statechart' s configuration.
ACTIVE	Verify if the states specified are active.
CONFIGURATION	Verify if the configuration specified is active

StatSim has a Programmed Execution Module (PEM) which comprises facilities to edit and execute programs written in ECL. During an execution (or simulation) the ECL interpreter accesses the data base of the statechart being simulated to make consistency checking. The same simulation procedure used in the interactive simulation is used to perform the actual simulation.

During a simulation the events generated by the probabilistic functions input to the interpreter which calls the simulation procedure. This procedure performs the actual simulation and returns the new configuration. It also handles all changes on the screen and in the log file. A report that shows the simulation steps is appended with new information at each step and a statistical report is generated at the end of the execution, based on

information collected and stored at each step. The user is able to create other reports based on control variables and the "MESSAGE" command.

## 5. Model Analysis

The statechart of Figure 1 shows two processors and a single queue with requests from processors being determined by the event **process**. If we consider the case that one of the processors **CONTROL\_PROCESSOR\_ONE** or **CONTROL\_PROCESSOR\_TWO** is free (either **FREE1** or **FREE2** is active) then either the transition labeled **process[cr(in(FREE1))]/request1** or **process[cr(in(FREE2))]/request2** will be fired leading to a configuration where one of the processors is busy (**BUSY1** or **BUSY2**).

When both processors are busy and **process** occurs then the transition labeled **process[cr(in(BUSY1) and in(BUSY2))]/x:=x+1** will be fired. When either **deliver1** or **deliver2** occurs and the queue is empty ( $x=0$ ), then a transition to **FREE1** or **FREE2** will occur. However, if the queue is not empty ( $x>0$ ) then the processor remains busy and the queue is decreased by one request, i.e., either the transition labeled **deliver1[cr(x>0)]/x:=x-1** or **deliver2[cr(x>0)]/x:=x-1** occurs. Note that both transitions can occur simultaneously, in which case the queue is decreased by two requests.

We show in Figure 3 two steps of a programmed execution comprising 100 steps where an error can be devised (the program is not shown, but it is similar to the one shown in Figure 5). In step 15 the two processors are busy and the queue is empty. The event **deliver2** occurs simultaneously with **process** and the system changes **BUSY2** to **FREE2** in step 16. Now, the event **process** causes the transition labeled **process[cr(in(FREE2))]/request2** to fire, but **request2** is lost because we cannot return to **BUSY2** in the orthogonal component **CONTROL\_PROCESSOR\_TWO** due to the statechart's semantics as a state change has already occurred in this step. Generation of the events **process** and **deliver2** in this step was the key to uncover this error. The statechart of Figure 1 is not correct as it cannot handle this case and other similar ones.

We show in Figure 4 a new version of the example shown in Figure 1 with a small change on its attending policy. A processor request is always immediately granted if at least one of the processors is free and the queue is empty, otherwise the request is queued. When the queue is empty and the request arrives at the same time step in which either one or both processors become free, the request is granted in the next time step.

```

Statechart in Simulation: CONTROL_PROCESS_WRONG
Number of Simulated Steps: 100
.
.
.
----- Step 15 -----
Configuration of Active States:

CONTROL_PROCESS_QUEUE
CONTROL_QUEUE
  QUEUE
CONTROL_PROCESSOR_ONE
  BUSY1
CONTROL_PROCESSOR_TWO
  BUSY2

Events triggered      :
Internal Transitions :
States Turned On     :
States Turned Off    :

Variable              Current Value      Average Value
x                      0                0.0000
y                      1                0.4000
z                      1                0.3333

Messages :
----- Step 16 -----
Configuration of Active States:

CONTROL_PROCESS_QUEUE
CONTROL_QUEUE
  QUEUE
CONTROL_PROCESSOR_ONE
  BUSY1
CONTROL_PROCESSOR_TWO
  FREE2

Events triggered      : deliver2,process,request2
Internal Transitions :
States Turned On     : FREE2
States Turned Off    : BUSY2

Variable              Current Value      Average Value
x                      0                0.0000
y                      1                0.4375
z                      1                0.3750

Messages :
----- Step 17 -----
.
.
.

```

**Figure 3 - Steps Report**

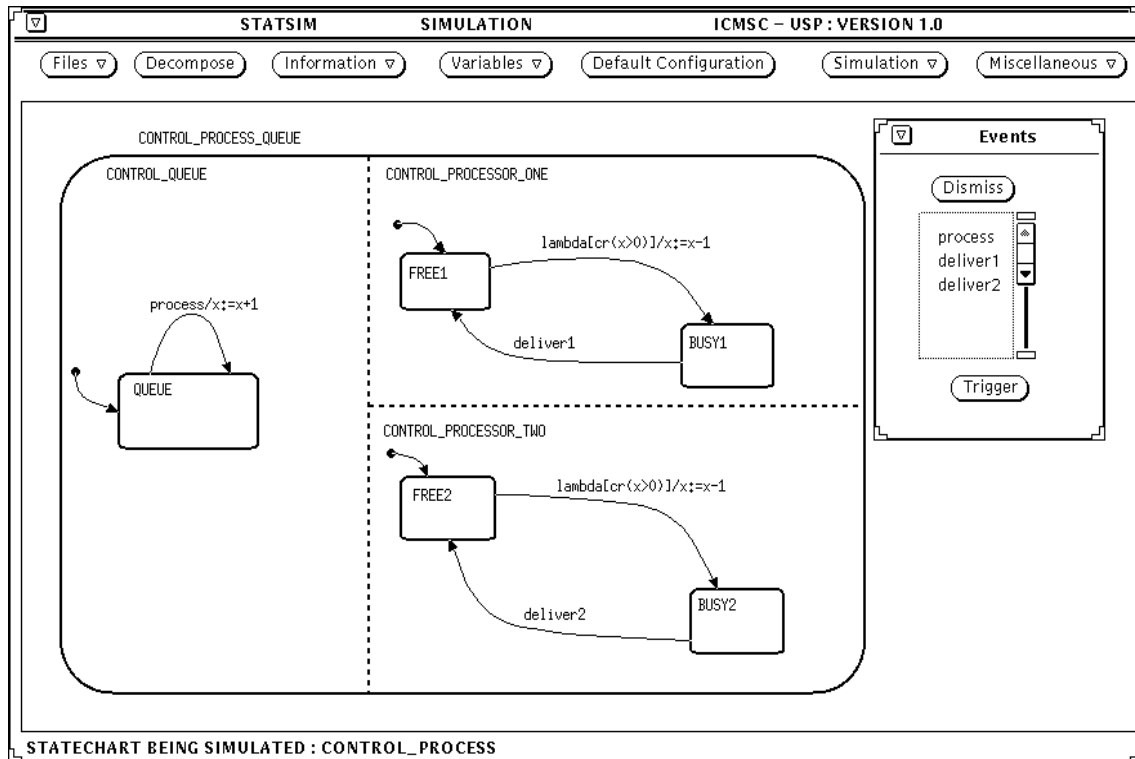


Figure 4 - Second Version

### 5.1. An ECL Program

We show in Figure 5 an ECL program created to simulate the second version of our solution (Figure 4). We can see that 100 steps are specified from the initial default configuration. A normal probabilistic distribution with parameters 2.5 and 0.5 was specified as the distribution of **process**. We also use two auxiliary variables *y* and *z*, to indicate when the processors are busy. When a **process** occurs and the queue is empty, it is incremented and one of the transitions **lambda[cr(x>0)]/x:=x-1** in the two orthogonal components **CONTROL\_PROCESSOR\_ONE** and **CONTROL\_PROCESSOR\_TWO** is nondeterministically fired and the queue remains empty.

The variables *y* and *z* are used in the ECL program of Figure 5 to verify if the processors are busy and, when this is the case, the ECL program schedules an event **deliver1** or **deliver2** for some steps ahead according to a randomly generated number, thus simulating the time this processor keeps busy. This program also verifies if the size of the queue becomes greater than three and prints the step in which this occurred and it also prints compulsorily this size after any twenty steps.

```

GLOBAL PARAMETERS

    NUMBER_OF_STEPS = 100
    INITIAL_CONFIGURATION = DEFAULT
    SIMULATION_RESULT = COMPLETE

DECLARATIONS

    EVENTS process NORMAL(2.5,0.5)

SIMULATION

    ALL STEPS
    BEGIN
        IF (x>=3) THEN
        BEGIN
            MESSAGE ("Number of process in queue >= 3")
            MESSAGE ("QUEUE = ", x)
        end

        IF (y = 1) THEN
        BEGIN
            EVENTS(deliver1,UNIF(5.0,1.0))
            y:=0
        END

        IF (z = 1) THEN
        BEGIN
            EVENTS(deliver2,UNIF(4.0,1.0))
            z:=0
        END
    END

    AT STEP 20, 40, 60, 80, 100
    BEGIN
        MESSAGE ("Queue Size = ",x)
    END

END_SIMULATION

```

**Figure 5 - Source Program in ECL**

## 5.2 - Statistical Report

A statistical report is produced as the result of the programmed execution. Figure 6 shows this report produced by the program in Figure 5, based on the statechart of Figure

4. The first two lines show the name of the statechart simulated and the number of steps of the simulation. It then shows information about the basic states, events, transitions and variables.

```
STATECHART: CONTROL_PROCESS
NUMBER OF STEPS: 100
```

---

STATE	PERCENTAGE OF STEPS IN WHICH WAS ACTIVE	FINAL SITUATION	AVERAGE NUMBER OF CONSECUTIVE STEPS IN THIS STATE
QUEUE	100.00	on	2.5789
FREE1	21.00	off	0.3125
BUSY1	79.00	on	4.1333
FREE2	20.00	off	0.1765
BUSY2	80.00	on	3.8125

---

EVENTS	NUMBER OF STEPS IN WHICH THIS EVENT HAPPENED	PERCENTAGE OF STEPS IN WHICH THIS EVENT WAS AVAILABLE
process	38	100.00
deliver1	15	100.00
deliver2	16	100.00

---

TRANSITIONS	NUMBER OF STEPS IN WHICH THIS TRANSITION WAS FIRED	NUMBER OF STEPS IN WHICH THIS TRANSITION WAS EVALUATED	RATE OF FIRING SUCCESS
deliver2	16	16	100.00
lambda[cr(x>0)]/x:=x-1	17	20	85.00
lambda[cr(x>0)]/x:=x-1	16	21	76.19
deliver1	15	15	100.00
process/x:=x+1	38	38	100.00

---

VARIABLE	AVERAGE VALUE	FINAL VALUE
x	2.3700	5
y	0.1600	0
z	0.1700	0

**Figure 6 - Statistical Report**

For each state it shows the state' activation rate, its final state (on or off) and the average number of consecutive steps that the system remained in this state. We can see

that **QUEUE** was always active and we can learn the average value of busy and idle times for the processors in this particular model, which are given in the third column. Processor one, for example, was busy 4.1 steps, in average.

Events are analyzed through the number of times they were generated and through the number of steps in which they were available. Transitions are analyzed through the number of times they were fired and evaluated. From these numbers we can obtain the rate of firing by evaluations. In our example we had 38 requests from processors of which processor one completed 15 and processor two completed 16. At the end 5 requests were enqueued and two were being attended (final situation of states **BUSY1** and **BUSY2** were **on**).

We also know that the transitions **process/x:=x+1**, **deliver1** and **deliver2**, must be fired every time they are evaluated, and this in fact happened. Both lambda transitions must be evaluated at all steps in which the states **FREE1** and **FREE2** are active and the condition [**x>0**] is true.

The statistical report shows the average and the final value of the model variables. In the example, variable *x* controls the queue' size and then we have the average number of processor requests enqueued in each step. These numbers have no meaning for *y* and *z*, used only as auxiliary variables within the program of Figure 5.

The statistical report provides essential statistical information about the examples and additional information about the statechart behavior. In examples for which the statistics are not important we still get a summary of the simulation and many verifications can be made, a transition which is always relevant and evaluated to true must have been fired at all steps. We can check, for example, if a transition which is always relevant and always evaluated to true has been fired at a 100% of the time steps or the internal events generated by a certain transition were generated the same number of times the transition was fired.

## 6 - Concluding Remarks

Execution of a model under randomic conditions complements other forms of validation as a wide range of scenarios not devised by the user can be generated. We have shown how this type of simulation can be conducted within the StatSim environment and how the statistical results can be interpreted and used to validate a model. We have also shown how to use programmed execution to reveal errors in a model, providing insight into ways for improving it.

Generating events based on probabilistic distributions facilitates simulation under many different situations: changing the distribution parameters or testing different combinations of distributions. In the example shown, we could simulate a critical situation increasing the rate of processor request and therefore the queue' size and also increasing the duration of processors. Situations in which the queue is almost always empty or in which a processor is much faster than the other one could also be easily modeled.

## 7 - References

- [1] BURNS, A.- Scheduling Hard Real Time Systems: a review. Software Engineering Journal, v.14, n. 3, 1991.
- [2] HUIZING, C.; ROEVER, W.C. - Introduction Design Choices in the Semantics of Statecharts. Information Processing Letters, v.37, 1991.
- [3] HAREL, D.; et al. - STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering, vol. 16, n. 4, 1990.
- [4] MARANINCHI, F. - Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by Using a Process Algebra. In: SIFAKIS, J. Automatic Verification Methods for Finite State Systems, Berlin, Springer, 1989. (Lectures Notes in Computer Science, 407).
- [5] HAREL, D. - STATECHARTS: a Visual Formalism for Complex Systems Science of Comp. Programming. v. 8, 1987.
- [6] HAREL, D. - Biting the Silver Bullet - Toward a Brighter Future for System Development. IEEE Computer, v. 25, n. 1, 1992.
- [7] LEVESON, N. C.; et al. - Experiences Using Statecharts for a System Requirement Specification. In Proc. Int Workshop on Software Specification and Design, Lake Como, Italy , 1991.
- [8] SOWMYA, A. - A Statechart-Based Specification and Verification of Real-time Scheduling Systems. In Proc. Int. Workshop on Real-Time Programming Bruges, Belgium, 1992.

- [9] WELLNER, P. D. - Statemaster: a UIMS Based on Statecharts for Prototyping and Target Implementation. in Proc. Human Factors in Computing Systems. CHI-89 Austin, Texas, USA, 1989.
- [10] MASIERO,P.C.; FORTES, R.P.M.; BATISTA,J.E.S. - Editing and Simulating Behavioral Aspects of Real Time Systems (in Portuguese), XVIII Integrated Hardware and Software Seminar, Santos, Brazil.. 1991.
- [11] SCHRIBER, T.J. - An Introduction to Simulation Using GPSS/H. USA, John Wiley, 1991.
- [12] BANKS, J.; CARSON,J.S. ; SY, J.N. - Getting Started with GPSS/H. Annandale, Wolverine Software Corporation, 1989.
- [13] CANGUSSU, J. W. L.; MASIERO, P.C. A Language for Programmed Execution of Statecharts (in Portuguese). In: XIX Integrated Hardware and Software Seminar, Rio de Janeiro, Brazil, 1992.
- [14] HAREL,D. et all - On the Formal Semantics of Statecharts, Proceedings of the 2nd IEEE Symposium on Logic in Computer Science, Ithaca, N.Y., 1987.
- [15] HOOMAN, J. J. M., RAMESH, S. Roever, W.P. - A Compositional Axiomatization of Statecharts. Theoretical Computer Science 101, 1992.
- [16] MASIERO, P.C.; MALDONADO, J.C; BOAVENTURA, I. A. G. - A Reachability Tree for Statecharts and Analysis of Some Properties. In Information and Software Technology, v. 36, n. 10, 1994.
- [17] CANGUSSU, J. W. L.; MASIERO, P. C. - Programmed Execution of Statecharts (in Portuguese). VII Brazilian Symposium of Software Engeneering. Rio de Janeiro, Brazil, 1993.