Interactive Timed Simulation of Distributed Systems - From PROMELA to PROMELA+

Marco Daniele, Paola Renditore, Roberto Manione

Via Borgaro, 21. 10148 Torino - ITALY
phone: +39-11-2285111; fax: +39-11-2286862
e-mail: manione@cselt.stet.it

Abstract *Simulation is a powerful tool in the development cycle of distributed systems; it has long been studied and practiced. The present paper presents an approach to timed interactive simulation based on the execution of timed CSP-like models; the language presented is PROMELA+, derived from PROMELA with the extension to temporal quantification.*

The whole simulation environment, named YES, includes also a statistical analyzer which requests to the executor as many samples as needed to generate the desired estimators with the given properties.

The YES simulation environment can be used for a number of purposes during the system development cycle, from evaluation of the specification to the performance forecasting of the system, to system functional and throughput stressing.

While the environment has been used for about two years, when a number of systems have been modelled in PROMELA+, work in progress tends to exploit the YES environment as a basic simulation environment: PROMELA+ is seen as a kind of intermediate language for the timed simulation of distributed systems: a number of compilers, which translate higher level distributed paradigms and notations into PROMELA+ for simulation, have been written; this approach saves the high effort of implementing and validating a whole simulator and allows to concentrate on the semantic aspects of the new language.

1. Introduction

Simulation is a major tool in the development cycle of large Hardware and Software distributed systems; the basic concept is to reproduce via simulation selected behaviours of the system under study. Depending upon the specific development phase, particular simulation flavours apply: as an example, during the *specification* phase the system specification can be executed in order to demonstrate its suitability to accomplish the desired tasks; during *system testing* the environment within which the real system will operate can be simulated: the so obtained traces can be used to stimulate the real system before its deployment in the field; finally in the *acceptance testing* the performances of the real system need to be evaluated: this can be achieved simulating environment configurations and workloads which may not be currently available.

A wide variety of application domains can take advance of such techniques, ranging from Network Management Systems, Network Elements, Switching Systems, Business Processes; their commonalities lay mainly in their reactiveness and distributedness.

Among the more common simulation objectives are:

- *functional evaluation* of the system specification;
- *performance estimation*: timed evaluation of the system specification;
- *real system functional testing*: functional simulation of the system working environment;
- *partial system simulation*: incremental testing of critical parts of the system;
- *system workload simulation*: performance stressing of the system ;

The above list leaves explicitly out the aspect of validation, as defined in [Holz91]; since the focus of this paper is mainly in system simulation, this valuable model checking technique will not be addressed here.

In order to have access to some or all of the above simulation features, the system under study and/or, to some extent, its working environment must be formally modelled; the PROMELA language has been chosen for a number of reasons, the more remarkable being:

- semantics close to the CSP
- clean syntax; easy to use orthogonal constructs
- already available functional simulator

On the PROMELA substrate, we decided to add new features: we extended the PROMELA language syntax and semantics and extended the *spin* simulator to support such additions; in order to distinguish our work from the original one, we called the extended language PROMELA+ and, consequently, the simulator, *spin*+; care was put in extending the language, in order to keep it

backward compatible: when the new features are not used, then the standard PROMELA primitives retain their original definition and semantics.

The *spin+* executor has been embedded into a complete simulation environment, named *YES (Yet another Event driven Simulator)*; the other major components of the environment are a statistical analyzer which processes the timed traces and generates the estimators for the desired variables and a plotter of the results. The YES environment has been used in the simulation of a number of real distributed systems, mainly for performance analysis purposes.

Furthermore, due to the power and flexibility of the PROMELA+ language, YES is being used as a mid-level language: some compilers from higher level modelling paradigms and languages for distributed systems have been built or are in progress. Such compilers translate higher level models into PROMELA+ models for simulation: this approach has proven to be more effective than implementing monolithic simulators of such high level languages.

Two examples of such strategy are: (1) a simulator of TINA-like Object Distributed specifications, [BBMMS95] [MaLa95] used within a CASE tool for the specification of Telecommunications Services and (2) a timed simulator on Event Trace Diagrams.

This paper will mainly deal with the basic YES system: the main extensions which brought the original *spin* to YES will be presented in the next paragraphs, §3 will deal with the extensions to the PROMELA language, §4 will present the statistical analyzer and §5 will deal with the interactive simulation environment; finally §6 will present simulation results.

2. Timed Simulation of Distributed Systems

Discrete Event simulation of distributed systems has long been studied and used; historically the first approach was to write ad-hoc programs which behaved like the real system as far as their external interface was observed, with respect to selected behaviours; a number of languages has been proposed in the past, with the special purpose of writing discrete event simulation programs for the timed simulation of systems, SIMSCRIPT and SIMULA [BDMN73] being only two of them; although such languages contained primitives and library functions supporting simulators development, such as random number generation, statistical analysis, events queue handling, etc... , considerable work had to be done in order to turn the conceptual model of the system into a running program yielding correct and useful results; furthermore such approach does not imply that the structure of the real system is reflected into the simulation program.

More recently, a number of simulation tools have emerged for the analysis and simulation of systems modelled high level description techniques, such as Queuing Networks and Petri Nets (in their various flavours: Stochastic, Generalized, Coloured, ...); some examples are: GreatSPN

[Chio87], TOPNET, Modline. These tools save the evaluator the task of writing and debugging a simulation program at the sacrifice of flexibility: for each tool only the considered types of entities and their respective semantics are available; some extensions to the basic notations have been proposed, mainly based on the possibility to attach procedural behaviours to the original primitives: unfortunately, many of the times, the resulting hybrid notation does not retain all the desirable properties of the original paradigm.

Queuing Networks allow to easily write models, sometimes at the cost of oversimplifying the original model; on the other hand, Petri Nets and the extended Generalized Stochastic Petri Nets (GSPN) lead to more realistic models at the expense of a higher complexity.

CSP-like languages, like PROMELA and their extensions bring the best of all the above approaches, offering a high flexibility in the development of the models without requiring programming: the user needs only to supply a model of the system; the model is then executed by the simulator, which produces the execution traces for further analysis.

3. YES: Timed simulation environment.

Most of the development effort presented in this paper has been directed to the development of a full simulation environment for interactive timed simulation. The resulting environment is called YES.

PROMELA, as accepted by *spin* version 2.0.4, has been taken as the basis for PROMELA+; a number of constructs have been added upon the original version, mainly to reach the goal of timed simulations; however, some features have been added at the purpose to further enhance the expressive power of the language. In the following we will describe in some detail the added features, leaving the details of the basic PROMELA constructs to [Holz91] and [Holz95].

The basic purpose of timed simulation of a model is to produce a timed trace from the execution of the model: the timed trace of the model execution can be taken as a sample of the timed trace of the real system to extent the model reproduces the real system.

A timed trace can be further analyzed in order to derive more aggregated informations, like average channel length, resource usages, etc; the PROMELA+ language and the YES environment provide a rich set of options to this purpose.

3.1 Basic concepts of timed simulation: time and resource.

The basic idea which led to the actual PROMELA+ language was to introduce temporal semantics to the PROMELA constructs (i.e. time spent in the execution of the instructions). The qualitative concept of time already implemented in PROMELA as the order of the actions during an execution is not enough to timed simulation.

In this perspective, an interpreter can be seen as an engine producing execution traces, where a trace is a sequence of state transitions, i.e. instructions.

spin provides the generation of such traces which we call *untimed traces*.

We want to obtain a *timed trace* as specified in [DaSc92], i.e. a sequence of state transitions partially ordered according to the times when they happen.

To fully comply with the system we need to model, we also need such traces to be *realistic* , i.e. we need to consider instruction durations and to take into account the fact that a real concurrent system is actually distributed over and carried on by a certain number of *resources*.

This comes quite natural if you consider the fact real systems are composed of different agents and that an action "consumes" time of the agent (the resource) which is actually executing it. This fact imposes a further constraint on the timed traces that can be produced by *spin*+, i.e. we can accept only those traces where the instructions executed by agents carried on by the same resource are totally ordered according to start times and durations.

In other words, this means that a total ordering of timed instructions has been imposed on instructions taken by a single resource, while partial ordering for instructions performed by different resouces is still possible in a valid timed trace.

Obviously, timed actions and resources make the simple random scheduling policy adopted by *spin* unsuitable: a more complex round-robin scheduling policy has been implemented which takes the notion of time into account.

Finally, in order to evaluate the time behaviour of a system, producing a timed trace of the execution of its model is not the final result: we need operators which filter and take measures out of such traces. This leads to the introduction of the observation primitive which allows to record selected system events and data for further statistical analysis.

We can now see in more detail how these concepts are mapped in PROMELA+.

3.2 Time specification: the Timed construct.

The `timed` construct allows to specify the time spent for executing a sequence of instructions. Instructions not enclosed within a `timed` construct are considered of null duration.

The present version of PROMELA+ supports the notion of time as a real quantity.

`timed` statements define sequences of statements to be executed with a transactional semantics (i.e. all or none, a blocking statement within a timed sequence causes a run-time error except if the blocking statement is the first one, in which case it acts as the guard of the transaction) taking a certain amount of resource time. Idle waiting is achieved using `sleep` as the only statement in an `timed` sequence; the `sleep` statement will cause the process to sleep for the specified amount of time without loading the resource.

The PROMELA+ syntax is:

```
timed '(' <delay> ')' '{' <statement_sequence> '}'
```

The <delay> option allows to specify a possibly stochastic duration for the sequence of statements enclosed in braces. At present, the following options for the <delay> qualifier are allowed:

- *uniform distribution*, denoted by `uni ':' <low>, <high>` where <low> and <high> are expressions evaluated to real numbers whose values represent the lower and the upper bound respectively for the duration value that will be generated at run-time;
- *normal distribution*, denoted by `gauss ':' <mean>, <std_deviation>`: in this case, the duration will be generated according to the normal distribution whose parameters (mean and standard deviation) are given by the two expressions <mean> and <std_deviation> specified in the clause;
- *negative exponential distribution*, denoted by `exp ':' <mean>`: the duration will be generated according to a negative exponential distribution characterized by the parameter 1/m, where m is the value of the <mean> expression, the mean value of exponential distribution;
- *constant value* denoted by `const ':' <expr>`: the duration is simply the value of <expr>.

3.3 Resource allocation: run statement and active proctype.

A PROMELA+ specification is executed in a distributed environment where a certatin amount of resources is available. At present the number of processors and the number of processes that can be simulated by *spin+* can be specified through a run-time option on the command line. This allows for flexible tailoring of the memory requirements of the simulator.

The language provides a simple way to map process instances on processors; this has been achieved by modifying the run construct as follows:

```
run <proc_type> '(' <actual_parameters> ')' on '(' <processor> ')'
```

This expression assigns the new instance of <proc_type> to the processor whose number is the result of the integer expression <processor>; the on clause can be omitted, in which case the process instance is assigned to a default processor.

Notice that <processor> expression is evaluated at run-time, allowing for sophisticated dynamic processor allocation.

The allocation facility is provided also for active proctypes; in this case a constant expression can be specified according to this sintax:

```
active'['<const1>']' on '('<const2>')' proctype '('<params>')' '{'<body>'}'
```

3.4 Observation constructs

During system execution, many different system parameters can be collected for further statistical analysis (performed by *Driver*, see § 4). Any arithmetic expression that can be written in PROMELA+ can be observed.

PROMELA+ allows related system parameters to be observed together in views. A view is a structured data type introduced via a `viewdef` declaration: `viewdef` allows to name the system parameters to be collected for the defined view type; the expressions defining such system parameters will be specified using the `observe` statement.

A short example can help in making some more details clearer.

```
viewdef view_type { obs usages[2], avg length }
view_type v;
...
observe _used(psr1)         at v.usages[0];
observe _used(psr2)         at v.usages[1];
observe len(channel1) at v.length;
...
```

This small fragment of PROMELA+ code shows definition of a new view template via the `viewdef` construct, the declaration of the view `v`, and a group of observations on `v`.

In our example, `usages[0]`, `usages[1]` and `length` are the parameters to be collected; the `viewdef` statement defines also the observation mode through the `avg` and `obs` qualifiers: the `obs` qualifier states that instantaneous values are observed, while `avg` specifies that a timed weighted average of the point values observed will be recorded for that system parameter. So, in our example, a generic sample collected for `length` is:

$$\frac{\sum_{i=1,..n} \text{len}_i(\text{channel } 1) * \Delta t_i}{\sum_{i=1,..n} \Delta t_i}$$

while, e.g. for `usages[0]` a generic sample is simply the value of `_used(psr1)`.

3.5 Enhancements in the expressive power of the language

Although PROMELA 2.0 has been significally extended with respect of the original PROMELA, extending the available data types, we have found that more data types would have been of help for clean and concise modelling of complex systems.. For this purpose, we have added a real data type, called `float`, which holds single precision floating point numbers (e.g. 32 bits wide); notice that this is useful to deal with time measures within models, being time a real

quantity. Moreover, the set of type constructors has been enlarged allowing multidimensional arrays.

On the control flow side, we have introduced weighted alternatives:

```
do (or if)
'::'     '['<expr_1>']' <sequence_1>
'::'     '['<expr_2>']' <sequence_2>
         ...
'::'     '['<expr_n>'] '<sequence_n>
od (or if)
```

where <expr_i> are expressions; they represent the weights assigned to the different choices; *spin+* will actually perform a pseudo random choice among those alternatives that can be executed at the time of the choice, giving probability proportional to the weight.

Since *spin+* can deal with time, the `timeout` construct has been extended to keep up with time; so, besided the standard `timeout`, the user can now specify

```
timeout after '(' <expr> ')'
```

meaning that if no event happens within <expr> time units of time, the `timeout` will be triggered and the process will be able to continue with the statement following the `timeout`.

This statement introduces the possibility to model systems whose behaviour explicitly depends on time, such as real time systems. Besides this statement, the following built-in functions and variables can be useful for the modelling of time-related behaviour: and for :

- `_work(<pid>)`: reports the total amount of time spent by process <pid> when executing instructions
- `_existing(<pid>)`: reports the total amount of time spent by process <pid> in the system
- `_used(<psr>)`: reports the total amount of time spent by processor <psr> executing instructions
- `_uptime(<psr>)`: reports the total amount of time since processor <psr> has been turned on
- `_sys_time`: reports the system absolute time reached by the system since the beginning of execution.

Such facilities can also be used for time measurement inside observations.

Moreover, we provide functions for the generation of numbers according to the following distributions:

- *uniform distribution*, denoted by `_uni '(' <low>, <high> ')'`
- *normal distribution*, denoted by `_gauss '(' <mean>, <std_deviation> ')'`
- *negative exponential distribution*, denoted by `_exp '(' <mean> ')'`

The meaning of the above function parameters is the same of the one introduced when speaking about `timed` contruct `<delay>` qualifier.

Finally, in order to easily control the execution, the `stop` statement has ben added:

```
stop '(' <expr> ')'
```

It stops simulation when `<expr>` returns a non null value.

3.6 Interpreter improvements.

The original *spin* version is rather concise and loose with respect to compile time checks; *spin*+ implements stronger type checking (both at compile time and at run-time) and stronger semantic checking.

Moreover, since the purpose of the YES simulation environment is to efficiently simulate large system, namely systems having hundreds and thousands of process instances distributed over nearly as many resources, the interpreter performance needed to be improved.

spin+ performance improvement started with the implementation of a more efficient mechanism of variable access; variable names are now resolved at compile time in terms of an offset within a local environment for local variables, or the global environment for global variables.

Furthermore, a new option can be used for those systems where the number of processes waiting to send and/or to receive on channels is quite large at any point in time; this new option allows to disregard such processes during the scheduling fase, thus reducing the number of active processes to be considered. This, with the removal of the constraint to delete from the system terminated processes starting from the one with the highest process id, allowed to obtain a dramatic improvement in performances. This brings the *spin*+ performances in the average of 15000 PROMELA+ instructions per CPU second on a SPARC Station 20 for both small and large systems.

3.7 Process scheduling policy

The scheduling algorithm for *spin*+ needs to deal with time and resources in a fair and realistic way, i.e. the scheduling policy needs to assure a system emulation as close to reality as possible, with time advancing as smoothly as possible over all the resources.

The scheduling policy presently adopted by *spin*+ is round-robin, based on the time of last service of processes: processes having the lowest last service time are considered first for execution.

Processes are given a slice of one instruction each time they are chosen for execution.

Furthermore, the scheduler takes care of instruction durations when scheduling processes, i.e. processes whose next instruction will be of null duration are given priority over those which will execute an instruction taking some amount of time.

To summarize, the scheduler will look on each processor for the lowest last service time in order to find out which instruction can be executed next; since the priority is given to null duration instructions, the control will be given to processes executing them; then, if no such processes exist, only one of the processes executing non null duration instructions will be given control for execution.

Such technique assures a fair and realistic development of the execution, not only in the long time average behaviour, but also at any point in time.

4. Driver: the YES statistical analyzer.

Since some kind of statistical behaviour is contained into a system specification (e.g. non-deterministic choices, statistical durations of actions, etc.), just running one execution experiment is not enough to obtain reliable figures about the time behaviour of such system: a suitable number of independent executions have to be run and the collected data (see § 3.4) have to be statistically analyzed: the *Driver* program performs this task.

Driver will determine the number of system executions and number of samples to be collected for each execution according to the statistical behaviour of a particular system parameter we call *system reference parameter*: intuitively, the higher the variance of such parameter is, the longer the executions will be.

First of all, a pilot execution is run; during such execution the statistical behviour of the system is evaluated and the number of samples to be collected in the subsequent runs is determined. Moreover, the number of samples to be discarded for the transient skipping is computed: such samples are those referred to the time needed for the system to reach a steady state.

A simulation allows to estimate the mean value of each system parameter using the multiple run technique [BDMN73]; each run produces a sample of such mean obtained by calculating the mean of the system parameter samples collected during the run. The means produced by the subsequent executions constituiting a simulation are combined to produce a mean of means. Given this last mean and its variance, a confidence interval of a requested confidence level is generated. The simulation is stopped when the radius of the confidence interval relative to the system reference parameter is sufficiently small, i.e. lower than a requested precision.

Output of the above statistical analysis is given both in textual and in graphical form, through Gnuplot.

5. Interactive functional simulation environment.

In the development of large hardware and software systems, an incremental process is welcome. This means that the development of the system goes from the stage where the system is completely prototyped using PROMELA+, through a possible series of intermediate stages where a part of the system has already been implemented while the remaing one is still simulated. At these intermediate stages, the two part of the system can be made interact to test the functionality of the implemented part. Of course, this process will lead to a completely developed system, which in turn can be finally tested against a model of its operating environment modelled using PROMELA+.

Such a simulation requires PROMELA+ models to interact with the outside; this needs communication facilities which have been implemented in PROMELA+: we call this facility *external channels*.

External channels are monodirectional and asynchronous; at present, we allow external channels only to be global, they cannot be passed as parameters and cannot be grouped into arrays.

From PROMELA+ point of view, an external channel can be used just like a "normal" channel, exept for its declaration that has the following syntax:

```
<direction>:= in | out
extern <direction> chan <name> = [slot_number] of { <message_types> }
```

A C-library for external channel handling (creation, reading, writing and closing) is provided for the development of the outside applications.

External channels are implemented via sockets, so that the partial system simulation and the remaining system can be run over different machines.

A command line option is provided in order to map external channels to the real port numbers associated to the sockets supporting them.

This mechanism allows to an external system to interact with a model of another system, but gives also the way to other interesting possibilities, such as driving the simulator behaviour from the outside, e.g to perform step-by-step simulation, debugging and so on.

6. Applications and results.

PROMELA+ is a clean and simple language whose semantics is well understood and whose interpreter *spin+* has been thoroughly tested.

This makes YES suitable for use as-is to model and evaluate systems both from the performance evaluation point of view and from the functional verification perspective. The small number of constructs present in the language makes infact the effort of learning the tool trivial, so that model prototyping can be done rather quickly.

Moreover, the expressiveness of constructs actually provided, the degree of reliability and efficiency presently reached by *spin+* makes it suitable for use as a basic simulation engine for the simulation of systems already specified using other description techniques.

In this last case, all that needs to be done, is the development of a suitable to-PROMELA+ parser. This task is actually simpler than the development of a new general purpose simulator, since tools are available to define efficient parsers and the process of validating the translation tool is trivially simpler than the one to test a whole simulator. Moreover, this validation task is made simpler by the simplicity of the language and by the relatively small number of primitives provided by the underlying language. Finally, this last approach saves the effort of learning a new description language when a description technique has already been adopted and consolidated.

Presently, we have been developing two such simulators by translation:

- simulator for ETD (Event Trace Diagrams): this application allows to model systems where a number of agents interact with each other through message interchange. Each agent is completely defined by the messages it can accept and by the messages it issues in respose of incoming messages. The tool allows to produce and evaluate the timed traces of such models.
- simulator of object oriented knowledge bases for interactive testing of diagnosis systems.

These two simulators will not be described in detail to give space to a detailed example of YES usage.

Among the cases actually analyzed using YES, there is a simple model of a menu application. We model the interaction of a user with an application providing a two level pop-up menu: we estimate the mean time spent for each entry in the menu at the two different levels.

The specification takes about 760 lines of PROMELA+ code and describes the user, taking different sessions with the application, the application itself and the observation instrumentation.

Fig. 1 shows the trace of the times spent for each level of menu (reply_time1 and reply_time2 respectively). Fig. 2 shows the evolution of the simulation with respect to the two estimated parameters.

We have applied the observation and estimation technique supported by the model language to *spin+* as well. This involved monitoring the *spin+* parameters of interest, such as the average number of PROMELA+ instructions performed per CPU second.

This approach has proved to be extremely useful to trim the inefficiencies of the interpreter and to tune the different improvement strategies (such as the descheduling of waiting processes we mentioned before).

An example of such results is shown in Fig. 3: the simulator can execute more then 10000 PROMELA+ instructions per CPU second (system execution creates about one hundred of process instances).

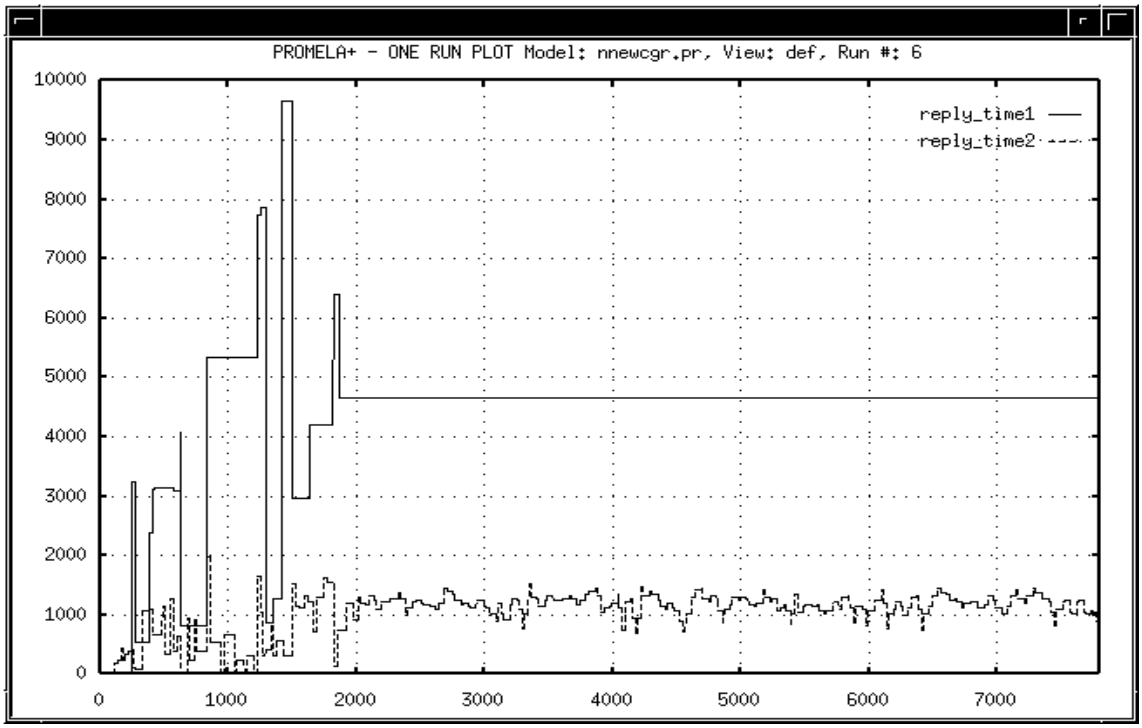

Fig. 1

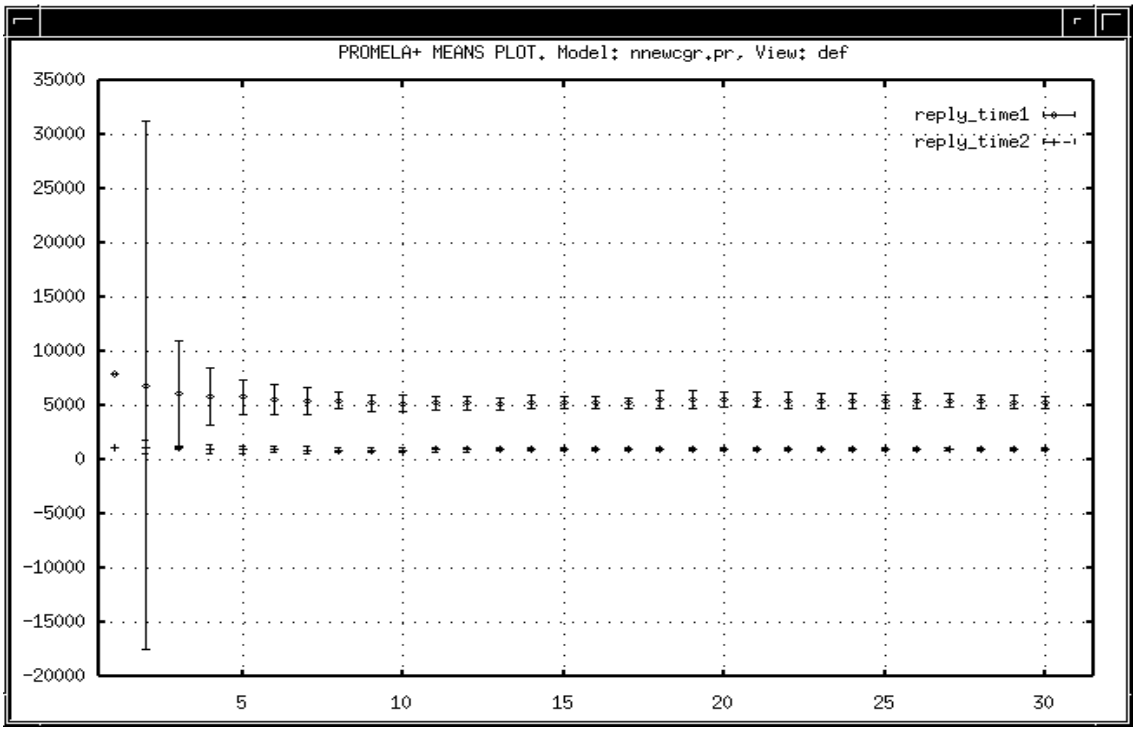
PROMELA+ MEANS PLOT. Model: nnewcgr.pr, View: def
reply_time1
reply_time2
35000
30000
25000
20000
15000
10000
5000
0
-5000
-10000
-15000
-20000
5
10
15
20
25
30

Fig. 2

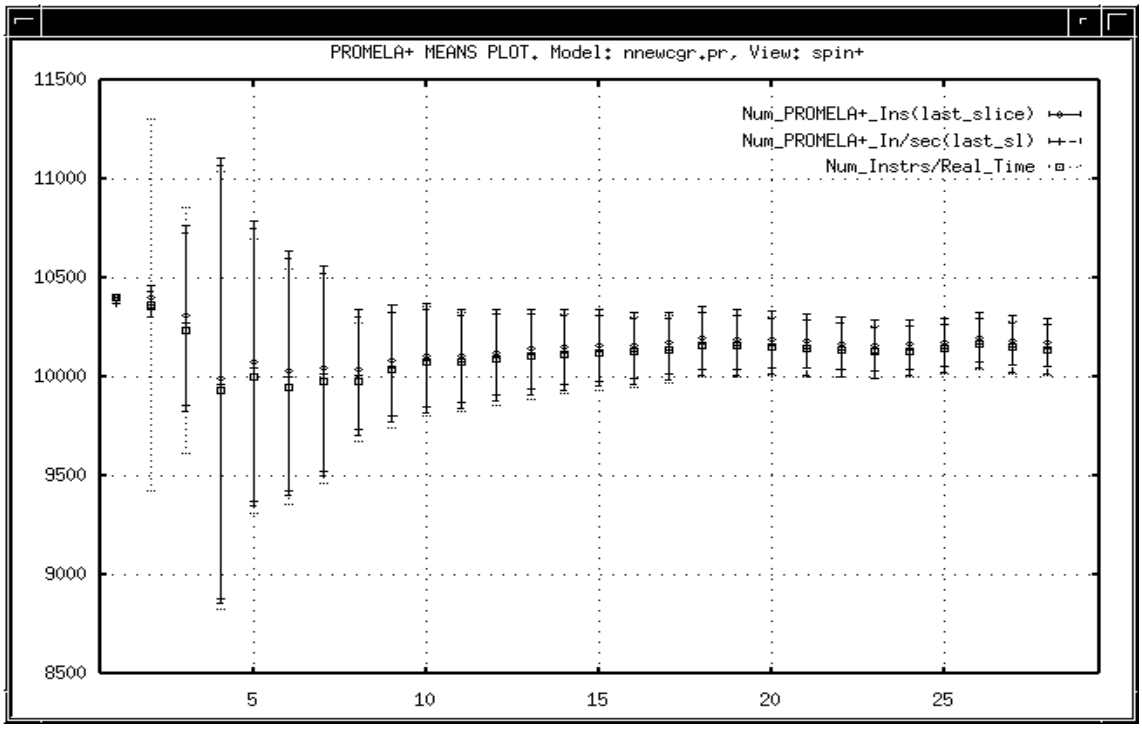
PROMELA+ MEANS PLOT. Model: nnewcgr.pr, View: spin+
Num_PROMELA+_Ins(last_slice)
Num_PROMELA+_In/sec(last_sl)
Num_Instrs/Real_Time
11500
11000
10500
10000
9500
9000
8500
5
10
15
20
25

Fig. 3

7. Conclusions and further improvements.

The introduction of a common language for system specification and timed simulation turned out to be extremely useful in the Telecommunications field, where formal specifications are studied within the development process and used for the a priori evaluation of systems.

Although the expressive power of PROMELA has significantly been enhanced to obtain PROMELA+ and inspite of its simplicity, we believe that the fact of being a textual language could be felt as a usability limitation by a user adopting PROMELA+ as is. To face this issue, a graphic editor is under development: it will allow the hierarchical editing of PROMELA+ specifications, retaining the conciseness of the original textual models, thus avoiding the major drawbacks we have experienced using other graphical description tools, such as e.g. GSPN, where, as the system size encreases, the readability of the model is highly compromized.

As for further enhancements that can be added to *spin+*, we think that an interesting possibility would be to give a measure of processor efficiency: if a processor is more powerful, the same instruction load could be carried out in a shorter time. Time associated with instructions, will then be interpreted as the "number of clock ticks" needed to perform the related task. This feature will add a further degree of flexibility in that the specification of agents involved in a system will not need to be modified (as it should be at present) to reflect a different hardware support to the system.

On the language side, a function definition construct would be useful; at present, infact, a function call can be substituted with a macro call or with a process creation; the first solution will result in heavier code while the second will cause an undesired load in the system interpretation. The definition of functions could be a cleaner solution.

As for the statistical analisys program, its effectiveness could be improved by adding the possibility to specify a number of system reference parameter instead of a single one; this will allow to run a single simulation in order to estimate, with the requested precision, all the required parameters.

Furthermore, the concept of interactive simulation could be extended to insert an interaction between the operator and the interpreter, thus giving the possibility for step-by-step simulation, trace, debugging features and on-line what-if analisys.

8. References

[BBMMS95] N. Bersia, P.G. Bosco, R. Manione, C. Moiso, M. Spinolo: A CASE Environment for TINA-Oriented Applications. ISS '95

[BDMN73] G. M. Birtwistle, O-J. Dahl, B. Myhrhaug, K. Nygaard: SIMULA Begin. Petrocelli-Charter 1973.

[Chio87] G. Chiola: A Graphical Petri Net Tool for Performance Analysis. Proc. 3rd. Int. Workshop on Modelling Techniques and Performance Evaluation, AFCET Paris March 1987.

[DaSc92] J.W. Davies, S.A. Schneider: A Brief History of Timed CSP. Technical Monograph PRG-96, University of Oxford (UK) 1992.

[Holz91] G. J. Holzmann: Design and Validation of Computer Protocols. Prentice-Hall International 1991.

[Holz95] G. J. Holzmann: What's New in SPIN Version 2.0 (Draft) 1995

[Hoar78] C. A. R. Hoare: Communicating Sequential Processes. Communications of ACM, Vol. 21, n. 8, pp. 666-677, Aug. 1978.

[MaLa95] R. Manione, A. Lagna: Simulation of Object-Oriented Distributed Systems via Compliation to Concurrent Process Level. 28th Annual Simulation Symposium, Phoenix April 1995.