Adding performance aspects to specification languages

Gregor v. Bochmann and Jean Vaucher
Departement d'Informatique et RO, Universite de Montreal,
CP 6128, Station "A", Montréal,
Canada H3C 3J7

Email: Bochmann@videx.udem.cdn

January 1988

Adding performance aspects to specification languages[6]

Gregor v. Bochmann and Jean Vaucher
Departement d'Informatique et RO,
Universite de Montreal, Canada

Email: Bochmann@videx.udem.cdn

Abstract

In the area of communication protocol design, so-called Formal Description Techniques (FDTs) are used to describe the behavior of the system components executing the communication protocol. Such formal specifications can be executed in a simulated mode in order to detect any remaining logical errors in the specification. If a specification language is extended with performance primitives which allow the description of such performance aspects as time delays, resource usage and stochastic behaviour, then simulation can be the basis for the performance evaluation of a complete system. The paper discusses this combination of logical and performance aspects in a single specification, and the choice of appropriate language elements for expressing the performance aspects. Such language elements are presented for several FDTs, namely Estelle, SDL and Lotos. Emphasis is placed on the justification of the chosen language elements and their relation with other well-known performance models, such Markov models, queuing networks, timed Petri nets and simulation.

1. Introduction

In the area of communication protocol design, so-called Formal Description Techniques (FDTs) are used to describe the behavior of systems. Estelle [Este 87], Lotos [Loto87] and SDL [SDL87] are formal specification languages which have been proposed as standards for the specification of OSI protocols and services [NBS 85]. SDL has also been used for the description of switching systems. The basic goal of such formal specifications is to ensure the correct specification and implementation of communication protocols. The formal nature of the specifications allows the application of partially automated methods for the validation of the specifications, for the implementation process, and for the systematic testing of resulting implementations [Boch 87c].

These formal specifications are intended to describe precisely the "logical" behaviour of systems, that is, the possible order of interactions and allowed parameter values of these. Most properties relating to performance aspects are, however, not addressed. A complete specification system should also address these questions. What is the maximum

---

[6] The work described here was funded by the Department of Communication of Canada through research contract OST83-0031 and by the Natural Sciences and Engineering Research Council Canada. Part of the notation described here was presented in 1984 to the ISO TC97/SC21/WG1 ad hoc group on FDT.

throughput of a link, what is the end-to-end delay on transmission, what degradation of service is introduced by a given error rate, etc...? Furthermore, protocols usually have many parameters which have to be tuned for optimum performance, such as duration of time-outs or number of buffers (window size) to be used. For these reasons, it would be useful to have operational specifications (simulations) that would allow the real-time behaviour to be measured. To do this, a specification language must include performance aspects which allow the description of such things as time delays, resource usage and stochastic behaviour.

The approach taken in this paper is to extend a given specification in order to address the performance question. This is in contrast to most traditional approaches where a new performance model is created in a different formalism to deal with performance issues. In that case, some of the "logical" properties of the system are often lost.

Estelle and SDL have many similarities, in particular, both use the concepts of Finite State Machines (FSM) to describe systems, but they also have important differences, most of which are related to the way system components can be created and interconnected. These languages allow certain performance elements to be specified. First, they have some basic means for talking about time. In the case of SDL, a global TIME variable is accessible and can be used for decisions and updating of variables. In the case of Estelle, so-called "delayed" transitions with minimum and maximum time limits can be defined. However, these primitives are insufficient for meaningful simulations.

The paper discusses the choice of appropriate language elements for expressing performance. Such language elements are presented for several FDT's, namely Estelle, SDL and Lotos. Emphasis is placed on the justification of the chosen language elements and their relation with other well-known performance models, such Markov models, queuing networks, timed Petri nets and simulation.The selection of language features for performance specification follows similar objectives as the selection of programming language features in general. On the one hand, one wants a small set of primitives with simple semantics (meaning) and easily interpreted, and on the other hand, one needs sufficient features to express the real system properties which are to be modelled. One therefore has to find the right compromise between these two objectives.

The paper is organized as follows. First, classical performance models are reviewed to bring out concepts useful in modelling performance. Then, the finite state machine model which serves as the basis for the existing specification languages is explained and two possible extensions to handle time duration are considered. Next, a complete set of performance primitives is proposed in the framework of an extension to Estelle. The use of these primitives is illustrated by the formal description of a simple network and its protocol. This is followed by a discussion of the application of similar primitives in the context of SDL and Lotos. We close with some comments on our experience with the use of an extended formal specification language.

2. Classical performance models

Models are often used to study the behaviour and performance of complex dynamic systems. For simple idealized systems, analytical models are appropriate; for more complex situations, one must resort to simulation. These models indicate what language features are useful to describe performance aspects in a formal specification language.

2.1 Analytical models

The following are classical models for that have been used to describe the performance of systems:

a) Markov models and probabilistic finite state machines (FSM):

> In such models, a system is characterized by a set of possible states in which the system can be and probabilistic state transitions that lead from state to state. Solving the model gives steady state probabilities of being in any given state.

b) Queuing models:

> These models are particularly suited for the description of the performance aspects resulting from shared resources. A specified system is characterized by a number of resources which process service requests. The execution of each service request takes a certain amount of time and each resource processes only one request at a time. When a resource is busy, further requests wait in a queue associated with the resource. Arrival of new requests and the service times may have random distributions.

c) Models with real time constraints:

> Some real-time systems require guaranteed response times for certain requests. To achieve this, these systems include so-called "time-outs" or timers. When started, timers will invoke some predefined action after a given "time-out" period, unless they are stopped by some other system activity. A simple model for this kind of behaviour is the "timed" Petri net [Merl 86] where the execution time for each operation has a minimum and maximum bound.

Each of performance models described above has a semantic which allows analytical modelling, at least for simple system descriptions. However, with real systems, often an exact analytical solution is not possible and simulation studies are required to obtain performance data.

2.2. Discrete event simulation

A **simulation** imitates system behaviour and system performance is estimated by measurements on the model. In **computer** simulation, the elements of a real system are represented by subroutines and records in a program in such a way that running the program produces results analogous to the behaviour of the system. In **discrete event** simulation, the activity of the entities in the system is viewed as a sequence of events (or instantaneous changes of state) separated by intervals of time. For instance, the loading of a truck would be modelled by a "start-load" event followed by a "stop-load" event after a delay representing the duration of the action. Parallel activity is imitated by interleaving the events of various entities and executing them in chronological order.

Simulation can be used to model systems of arbitrary complexity and size. However, simulation is expensive and simulation methods only provide approximate solutions (the

more precise a solution is sought, the more computer time is required). Often, simulation is not used to get precise estimates of system performance; rather simulation is used to get understanding of the system and to identify bottle-necks. Once a system is understood, the simulation can be discarded and performance obtained from simple analytical models of the identified bottle-necks.

GPSS, one of the oldest simulation languages, introduced many concepts that are useful in modelling [Schr 74]. GPSS conceives reality in terms of transactions (processes) moving through a system and requesting the use of resources. The passage of time is modelled by an *advance dt* primitive. Transaction use a *seize* operation to try and obtain the resources they need and they are blocked if the resources are busy. They *hold* them for a given *service* time and *release* them to be used by the next transaction. GPSS also has facilities to analyze performance: statistics pertaining to all resources are gathered automatically and transit times through various parts of the systems can be measured. The concepts of processes, resources and time advance are all very pertinent to protocol specification languages.

3. Finite State Machines

Several formal description techniques are based on finite state machines (FSM). We consider here a FSM with one or several input and output streams, as shown in Figure 1. Each FSM is characterized by a finite set of internal states and sets of possible inputs or outputs for each stream. Two kinds of transitions are considered: (a) An input transition consumes a particular input interaction from a particular input stream; it can only be executed if the given kind of input is at the head of the given stream and the machine is in a particular state. (b) A spontaneous transition consumes no input; it can be executed if the machine is in a particular state. Both kinds of transitions lead to a new state and may produce output over one or several output streams.

Figure 1b shows possible transitions for the machine of Figure 1a. There are three possible states S1, S2 and S3. In the notation used here, "A:IN / B:OUT" means that a transition requires the input "A" to be present at the head of the input stream "IN" and as a result of the transition, "B" will be output on stream "OUT". In the example, there are 2 input streams IN1 and IN2 and one OUTput stream. The transition from S1 to S2 as well as that from S3 back to S1 both require "a" to be present at the head of "IN1". The transition from S2 to S1 requires a "b" on stream "IN2". The transition from S2 to S3 is spontaneous, consuming no input. The transition from S1 to S2 produces "b" on the output stream; the one from S2 to S1 produces an "a" and none of the other transitions produces any output.

A set of FSMs becomes a system of interconnected FSMs if some of the output streams are identified or connected with some of the input streams.

In the specification language Estelle, the machine of Figure 1 would be described as a **module**:

```
module M1_type;
        ip      In1, In2, Out ;
        state   S1, S2, S3 ;
```

```
        trans from S1 to S2 when IN1 . a
              begin output  OUT.b  end;

        trans from S2 to S1 when IN2 . b  ;

        trans from S2 to S3
              begin  output  OUT.a  end;

        trans from S3 to S1 when IN1 . a
              begin     end ;
end;
```

First, the possible states and the *interaction points* (streams) are declared. Each transition is described by a **trans** statement with **from** and **to** clauses indicating initial and final states. A transition which is triggered by availability of input has a **when** clause; instantaneous transitions have none. Finally, output statements specify the creation of messages. The notation for IO is "**interaction_point . message**". The notation used above is based on that of Estelle and it will be extended to include performance aspects.

A set of FSMs becomes a system of interconnected FSMs if some of the output streams are identified or connected with some of the input streams. For example, this would occur if the output stream OUT of M1 in Figure 2 were to be connected to the input of another FSM. Then an output as a result of a transition in M1 could trigger a further transition in the connected machine. A possible notation for the creation and interconnection of the FSMs in the example could be:

```
M1, M2 :  M1_type;

connect M1. OUT to M2.In2 ;
…etc…
```

3.1 Performance models for interconnected FSMs:

The first step in specifying performance is representing the passage of time. There seems to be basically two ways in which an FSM model can be extended to do this: with Markov transitions or *lengthy* transitions.

(1) Markov performance model for FSMs:

Here a transition is instantaneous but some time elapses between the instant when it *could* occur and that when it *does* occur. For each transition t of the machine, a distribution function P_t (T) defines the probability that the machine does this transition within T time units after the transition has become possible, and assuming that no other transition has been executed.

(2) Transition execution performance model:

Here, transitions start as soon as possible. However, a transition takes some amount of time and the execution of one blocks the execution of others. To deal with the case when several transitions become possible at the same time, each transition is assigned a

probability and one of the possible transitions is selected at random according to the probabilities. The transition execution time may be a random variable, where a distribution function S_t (T) indicates the probability that the execution of the transition t will terminate within T units of time.

The first model is conceptually simpler. The second model has the advantage that it can be used to naturally model shared resources with FIFO queuing of requests. The requests wait in the input stream until they are processed, and the processing is modelled by the transition and its execution time. Only one request (transition) is processed at a time by any given FSM. Various other performance models have been described in the literature, e.g. [Moll82] and [Krit 86].

4. Performance models for Estelle/SDL

Estelle and SDL can be considered as extensions of the "interconnected FSM model" described previously. The extensions are related to the definition of input/output parameters, local variables (in addition to the STATE variable which identifies the FSM state), data types and procedures/functions for defining the transition operations in more detail. The two languages have many similarities, in particular the basic state transition model, but also important differences, most of which are related to the way component modules are created and interconnected.

4.1. An example

To illustrate the use of performance description and motivate the proposed extensions, we shall consider the specification of the simple system shown in Figure 2. A formal description of this system is given figure 3. In the system, several *users* are inter-connected via a *Network Service Provider*. The users send messages at random intervals to other users. Several typical situations will be considered such as receiving messages and responding with acknowledgements as well as retransmitting messages if no acknowledgement arrives within a specified *time-out* period. The description will express transmission delays, the possibility of message loss and the maximum throughput capacity of the user links.

In order to focus on the language aspects relevant to the present discussion, many details in both the model and the description are omitted: the messages have neither headers nor content, message recipients are chosen at random and some declarations and initializations will be missing. We describe first the aspects of the specification which do not concern performance. The performance aspects will be discussed later when the relevant specification primitives are introduced.

The specification (figure 3) start by declaring the message types (line 1) that can be exchanged over channels (line 2) between the modules in the system. Essentially, the system will be composed of one "NS_provider" module whose description starts at line 3 and several "users" described in lines 11-19.
The actual creation of the modules and their connection is not shown. During this phase, the relevant "interaction points" or **ip** defined at lines 4 and 12 will be linked with statements such as:

connect U1.Inq **to** Network.NS_out [1] ;
connect U1.Outq **to** Network.NS_in [1] ;

The NS_provider module defines three transitions (lines 8,9,10). The first two deal with reception of a message from a user over a NS_in port: in the first case (line 8), the message is sent out over the network, in the second case (line 9), nothing is sent out and there is loss of the message. The expression in the **when** clause of a transition serves 2 purposes. First, it identifies the stream and message type that will trigger the transaction. Secondly, it allows extraction of information from the message into local variables. For the transactions considered, *addr* will be assigned the **ip** address on which the message was received and *kind* will be set to the actual parameter of the message. In Prolog parlance, one could consider the operation as a *unification* between the message and the **when** clause. The final transition (line 10) passes on a received message to the correct user.

The user module defines two states *basic* and *waiting* (line 13) and four transitions (lines 16,17,18,19). The machine is in the *waiting* state after it has sent out a message and until an acknowledgement is received; otherwise, it is in the *basic* state. The first transition (line 16) handles the reception of unsolicited messages. These are acknowledged. The next transition (line 17) generates messages spontaneously. The third transition treats acknowledgements returning to the *basic* state. The final transition specifies that messages are retransmitted if no acknowledgement is received within a time-out period.

4.2. Performance parameters for Estelle specifications

This section defines some extensions to Estelle for defining performance parameters of specifications. These extensions are mainly based on the transition execution performance model described earlier.

4.2.1. Resources

An instance of an Estelle module is considered a resource. Input interactions arriving at an interaction point of the module enter a "common" input queue or an individual queue associated with the particular interaction point. The selection of the next transition to be executed is assumed to take no time. During the execution of a transition the module resource is held and no other transition may be performed by the same module. The outputs generated by the transition are available at the end of the transition. The execution time of a transition is indicated by a HOLD clause (extension of Estelle) of the form

hold for <expression>

where <expression> is an real value expression in time units.

For certain applications, it was found convenient to introduce the concept of declared resources. In this case the HOLD clause of the transition has the form

hold <resource> **for** <expression>

and <resource> is a variable access expression referring to a variable of type resource. The performance semantics of this clause is as follows: the transition has an additional

enabling condition, which requires that the <resource> must be free. If and when the execution of the transition is decided, the transition is executed in zero time and the outputs are produced; however, the resource remains occupied the amount of time specified by <expression>. It is therefore possible that immediately after the execution of the transition, another transition associated with another resource could execute, while a transition associated with the same resource must wait.

For instance, the Network module defined in Figure 3 receives input packets over a number of interaction points. In order to model the maximum throughput available for a given interaction point, a corresponding resource (IP_resource[i]) is declared within the module (line 6) and the input transitions receiving a packet hold the corresponding resource for the time proportional to the length of the packet received (lines 8,9). The reception of packets over different interaction points may proceed in parallel.

If the resource exists in a certain number of identical units, it may be convenient to indicate for a given transition how many units of the resource are required for the execution of that transition. This may be expressed by the notation

hold <number of units> **units** <resource> **for** <expression>.

4.2.2. Transition probabilities

As discussed in Section 3.1, it is sometimes necessary to indicate with which probability the different transitions which are possible in a given system state will be executed. Due to the extensions that Estelle provides in respect to the simple FSM model, a transition, in Estelle, has parameters: they include the parameters of the input (if any) and the present values of the local variables. These transition parameters may influence whether the transition is possible. In addition, if executed, they may also influence the values of output parameters and updated variables. The present (FSM) STATE and available kinds of inputs at the heads of the input streams do not completely determine which transitions are possible. Therefore it is not clear how transition execution probabilities (as in the transition execution performance model) can be associated with the transitions in a straightforward manner.

Since a given transition may "compete" with different sets of other transitions depending on the available inputs and the module state, its probability of execution may be specified indirectly by assigning a WEIGHT to the transition through a clause of the form

weight (<expression>)

where <expression> is a real value expression. The semantics of this clause is that the probability of selection of this transition for a particular system state is equal to the value of this <expression> divided by the sum of the weights of all transitions enabled in that system state.

In the example, **weight** clauses (lines 8,9) are used to indicate that on the average one message out of 1000 is lost .

4.2.3. Interaction queues with transmission delays

For modelling the transmission delays in telecommunication networks, it is convenient to introduce transmission delays for input/output streams. A similar approach is often taken in reachability analysis for protocol design validation where ad hoc models are used for the communication medium between the two communicating protocol entities. Properties such as FIFO discipline, and transmission error and loss possibilities are important not only for the performance but also for the logical aspects of protocol operation.

The basic performance parameters of a transmission medium are the delay and maximum throughput. The latter can be modelled by associating a resource with the input to the medium (sect 4.2.1.). Its service time will limit the number of transmission requests that can be handled. For the description of transmission delays, an extension to Estelle is introduced by which additional properties can be defined for the input queue associated with a given interaction point. The syntax of the interaction point declaration becomes

<interaction point> <properties> ":" <interaction point type>

where <properties> can be of the forms

delay <expression>

or

fifo delay <expression>.

The meaning of the first form is that an output generated for the given interaction point is delayed by the amount specified by <expression> before it is entered into the input queue of the interaction point. In the case of a constant <expression> the implicit FIFO property of the Estelle queues remain valid. However, if the <expression> contains random distribution functions, the order of arrival of interactions in the queue may be different from the order in which the outputs where generated. In other words, some interactions may overtake others. When the second form of the <properties> is used the delays will be lengthened, if necessary, in order to maintain FIFO order.

In the example of figure 3, transmission delays are modelled by using a FIFO *transit_queue*. Incoming messages are not sent out immediately to their receivers; rather, they are first placed in the transit_queue which has been declared to operate in FIFO mode with normally distributed random delays (line 5). Only on exit from the transit_queue are the messages placed on outgoing streams through the transition of line 10.

4.2.4. The DELAY clause

The DELAY clause for spontaneous transitions is already defined in Estelle. The proposal here is slightly different. It is assorted with an enabling condition of the form:

provided <enabling_condition>
delay <expression>

where <expression> is a real value expression in time units. The semantics of this clause is as follows: the transition is scheduled for execution when its <enabling_condition> has been satisfied for at least <expression> time (if transitions are executed during this time interval, the condition must remain true in between the transition executions). An example

is shown at line 17 for the transition which generates the messages coming into the system and at line 19 with a constant delay to model a time-out.

4.2.5. Use of random distribution functions

It is important to note that probability distributions may be used for describing non-deterministic behavior of the specified module. The simplest notation for such distributions seems to be the use of pseudo-random functions that return (random) values which have a given distribution. For simulation studies, it is important to allow for the use of independent streams of random numbers. Random functions are used at line 5 to specify transmission delays and at line 17 to compute intervals between spontaneous incoming messages.

4.3. Performance parameters in SDL

As mentioned above, SDL and Estelle are similar in many aspects. The following discussion indicates to what extent the same performance concepts can be used in the context of SDL.

Resources: An SDL process instance corresponds to a module instance in Estelle. An SDL transition corresponds to all Estelle transitions for a given STATE and type of input. Within an SDL transition, different cases (possibly depending on input parameters) may be considered. Like in Estelle, a resource may be associated with a process which can be held during a transition. However, the declaration of multiple resources seems to be less useful, since an SDL process has only a single common input queue (while an Estelle module instance may have individual input queues for all its interaction points). Therefore the parallelism in the system of Figure 3 cannot be directly obtained in the SDL context.

Transition probabilities: SDL has no possibility for implicit non-determinism, and therefore there is always at most one SDL transition to be executed. Different probabilities for different branches of execution can, however, be introduced by defining decisions which may depend on random functions, or which are not completely defined, leaving thus room for different decision outcomes. This is similar to sequential programming languages where the conditions used in IF or CASE statements may not be deterministic. Instead of introducing transition probabilities, like in Estelle, it may therefore be useful to introduce the possibility of non-deterministic decisions with probabilities for each of the possible decision outcomes.

Transmission delays: The same concept as for Estelle could be used.

DELAY clause: SDL does not have such a construct. Instead, as mentioned previously, a global TIME variable can be read, and its value can be used to influence the system behavior.

5. Performance parameters in Lotos/CCS

5.1. Short introduction to Lotos

In contrast to SDL and Estelle, Lotos uses rendezvous interactions. The rules for the sequential ordering of interactions in Lotos are largely based on CCS [Miln 80]; however, more than two processes may participate in a single rendezvous interaction. In Lotos, there are no implicit queues associated with interaction points. The Lotos "gates" play the role of interaction points, and an interaction at a gate can only be executed if all Lotos "processes" coupled to the gate are ready for that interaction. For example the process *simple* defined below uses the gates a, b, x, and y for its interactions.

```
process simple [a, b, c, x, y] : noexit :=
  y ; (   a ; suite_a [x, y]
      [] b ; suite_b [x, y]
      [] i ; suite_c [x, y]
      [] i ; suite_d [x, y]
      [] i ; x ; simple [a, b, x, y]  )
endprocess
```

The body of this process definition indicates that the simple process will first execute the interaction y and then may either execute in rendezvous with its environment the interactions a or b, in which cases it will continue with the behavior defined by suite_a or suite_b, respectively (the "suite" behaviors will only involve the visible interactions x and y), or it will make an internal transition, indicated the action i. In the case that it choses the last alternative, the process will execute the interaction x and thereafter start again with the interaction y.

Lotos also has facilities for defining data types as well as process and interaction parameters. Algebraic data type definitions can be written, similar to [ACT ONE]. The notation for interaction parameters is similar to CSP [Hoar 78] and not further explained here.

5.2. Performance parameters

It seems only two performance concepts are sufficient in Lotos to express most practical performance questions. These concepts correspond to the execution time of transitions and transition probabilities. The following notation could be used. The notation

wait <expression>

can be associated with an internal action "i" and means that the interaction requires the time period specified by the <expression>. A similar notation has also been used in [Quem 87]. The notation

weight <expression>

can be associated with an internal action which introduces an alternative of a choice. The <expression> defines the weight of that alternative (similar as described in Section 4.2.2) among all those alternatives that start with an internal action "i". For example the behavior expression

```
(    a;            suite_a [x, y]
  [] b; i wait 50; suite_b [x, y]
```

```
[] i weight 2;   suite_c [x, y]
[] i weight 1;   suite_d [x, y]
[] i weight 1  wait 100; x ; simple [a, b, x, y] )
```

defines a process which may participate in actions a or b (depending on its environment) or may choose one of the last three alternatives. Among the cases that one of the latter are chosen, suite-c will be executed with probability 1/2, suite_d and suite_e with probability 1/4. In case that the last alternative is chosen, a delay of 100 units is introduced before the execution of the simple process starts again. A delay is also introduced if action b is executed, such that suite_b can only start 50 time units later. In the other cases, the subsequent actions would start immediately provided, however, that the environment of the process does not introduce additional delays.

These basic performance primitives can be used, together with the normal features of the Lotos language, to construct processes that behave like resources with queuing delays or like communication media, as shown below. Therefore the above basic performance features seem to suffice for most typical applications.

A resource which remains reserved for t time units can be written as a process of the form

```
process resource [G] : noexit  :=
    G ; i wait t ; resource [G]  endprocess
```

which participates in the action G, then waits t time units and starts again. It can be used to limit the speed of execution of a very fast process executing interactions at the gate a by invoking the resource process in parallel, coupled with the former. This can be written as

```
very_fast_process [a]  ||  resource [a]
```

A transmission medium with random transmission delay can be written as

```
process medium [In, Out] : noexit :=
  hide Middle in
    delay [In, Middle]  ||  queue [Middle, Out]
endprocess
```

where *queue* is defined as a normal FIFO queue; In and Out are the gates where the messages are entered into the medium and received, respectively. The gate Middle is externally not visible and is used to transfer the messages from the process delay to the process *queue*. The latter keeps the messages in FIFO order until the user gets them. The process *delay* may be defined as follows

```
process delay [In, Out] : noexit  :=
  delay_a_message [In, Out]  |||  delay [In, Out]
  where process delay_a_message [In, Out] : noexit :=
        In ?x:message ; i wait <expression> ;
        Out !x; stop  endprocess
endprocess
```

This definition shows that a *delay_a_message* process instance is available for each message that is entered. The process waits a specified delay and then presents it at its Out

gate. This gate is in fact the Middle gate through which the message is entered into the queue process and available for the user.

In the case that the medium loses messages occasionally, the *delay* process body could be defined by the body

```
In ?x:message ;
(   (i weight 99 wait <expression> ; Out !x )
 [] i weight 1 (* loss *)  )
; stop  endprocess
```

5.3. The Network example

Using the concepts introduced in Section 5.2, it is not difficult to write a Lotos specification of the Network example discussed in Section 4. Figure 4 gives the definition of the user process and Figure 5 shows the interconnection of the different system parts, similar to the structure given in Figure 2. This specification is believed to be equivalent to the one given in Figure 3, not only concerning the logical behavior of the system, but also for its performance aspects.

The definition of Figure 4 indicates that the user process remains in the basic state until a normal message is output (first line of body definition). The second line of the definition introduces a delay for this output to occur. In the basic state, messages that are not of type "acknowledgement" are acknowledged. If the process receives an acknowledgement in the waiting state, it goes back to the initial state; however, after a time-out delay it will send a "retransmission" message.

6. Discussion

The performance concepts described above are closely related to the performance models of simulation languages such as GPSS [Schr 74] and Simula [Dahl 71]. The concepts have, however, been adapted to the particular context of the FDT's used for the description of communication systems. Similar approaches can be used for adding performance aspects to other specification languages.

The characteristic feature of the performance extensions to the FDTs described here is the possibility of combining the analysis of logical correctness of a specification with the evaluation of its performance. A case study has been done for the OSI class 0/2/4 Transport protocol [Boch 87e]. The same Estelle specification of the protocol was used for both the simulation and a semi-automatic implementation [Boch 87i]. For the simulation studies, an Estelle compiler generated Pascal code that was linked to a simulation package also written in Pascal [Vauch84b].

User processes and an underlying Network service similar (but more complex) to Figure 3 were also written in Estelle to provide an environment in which the Transport protocol processes could be simulated. Simulation runs were compared to the real Transport protocol running on our VAX-VMS environment, and all experimental results could be reproduced. These simulations were useful for several reasons:

(a) Some errors in the specification (and therefore in the implementation) were found and corrected.
(b) The simulation showed that the performance bottleneck was CPU usage; this had not been expected.
(c) We could find optimal values for certain protocol parameters, like the number of credits allowed for each user and some retransmission time-outs.

Our experience has shown that the extensions described here are both practical and useful. The simulations helped to improve both the reliability and the performance of protocol implementations.

7. References

[ACT ONE] H.Ehrig and B.Mahr, Fundamentals of Algebraic Specifications 1, Springer Verlag, 1985.

[Boch 87c] G.v.Bochmann, "Usage of protocol development tools: the results of a survey" (invited paper), 7-th IFIP Symposium on Protocol Specification, Testing and Verification, Zurich, May 1987.

[Boch 87e] G.v.Bochmann, D.Ouimet and J.Vaucher, "Simulation for validating performance and correctness of communication protocols", Tech. Report, Department d'IRO, Universite de Montreal, 1987.

[Boch 87i] G.v.Bochmann, "Semi-automatic implementation of Transport and Session protocols", Computer Standards and Interfaces 5 (1987), pp. 343-349.

[Dahl 71] O-J. Dahl, B. Myhrhaug and K. Nygaard, "SIMULA Common Base", Publ. Norwegian Computing Center, Blindern, Oslo (1971).

[Este 87] ISO DIS9074 (1987) "Estelle: A formal description technique based on an extended state transition model".

[Hoar 78] C.A.R. Hoare, "Communicating sequential processes", Comm. ACM 21, 8 (Aug. 1978), pp. 666-677.

[Krit 86] P.S.Kritzinger, "A Performance Model of the OSI Communications Architecture", , IEEE Trans. on Communications, Vol. Com-34, No. 6 (June 1986), pp. 554-563.

[Loto 87] ISO DIS8807 (1987), "LOTOS: a formal description technique".

[Merl 76] P.M.Merlin and D.J.Farber, "Recoverability of communication protocols - Implication of a theoretical study", IEEE Trans. on Communications, Vol. Com-24 (Sept. 1976), pp. 1036-1043.

[Miln 80] R.Milner, "A calculus of communicating systems", Lecture Notes in CS, No. 92, Springer Verlag, 1980.

[Moll 82] M.K. Molloy, "Performance analysis using stochastic Petri Nets", IEEE Trans. on Computers, vol. C31, pp.913-917, 1982.

[Quem 87] J.Quemada and A. Fernandez, "Introduction of quantitative relative time into Lotos", Proc. Specification, Testing and Verification of Communication Protocols, VII (IFIP), North Holland Publ. 1987.

[SDL 87] CCITT SG XI, Recommendation Z.100 (1987)

[Schr 74] T.R.Schreiber, "Simulation using GPSS", Wiley & Sons (1974).

[Vauc 84b] J. Vaucher, "Process-oriented simulation in standard Pascal", Proceedings of the Conference on Simulation in Strongly Typed Languages, San Diego, February 1984.

[Viss 86] C.Vissers, "Formal description techniques for OSI", Proc. IFIP Congress 1986, Dublin.

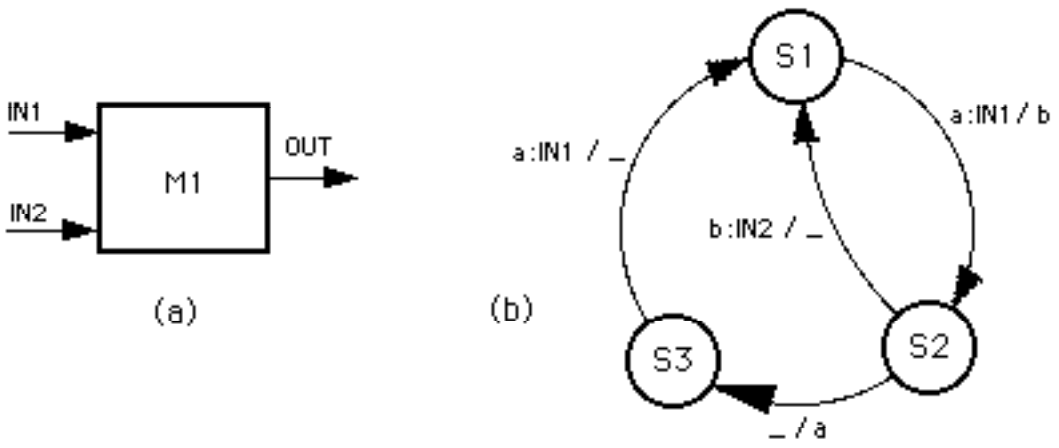

Figure 1: FSM with several IO streams

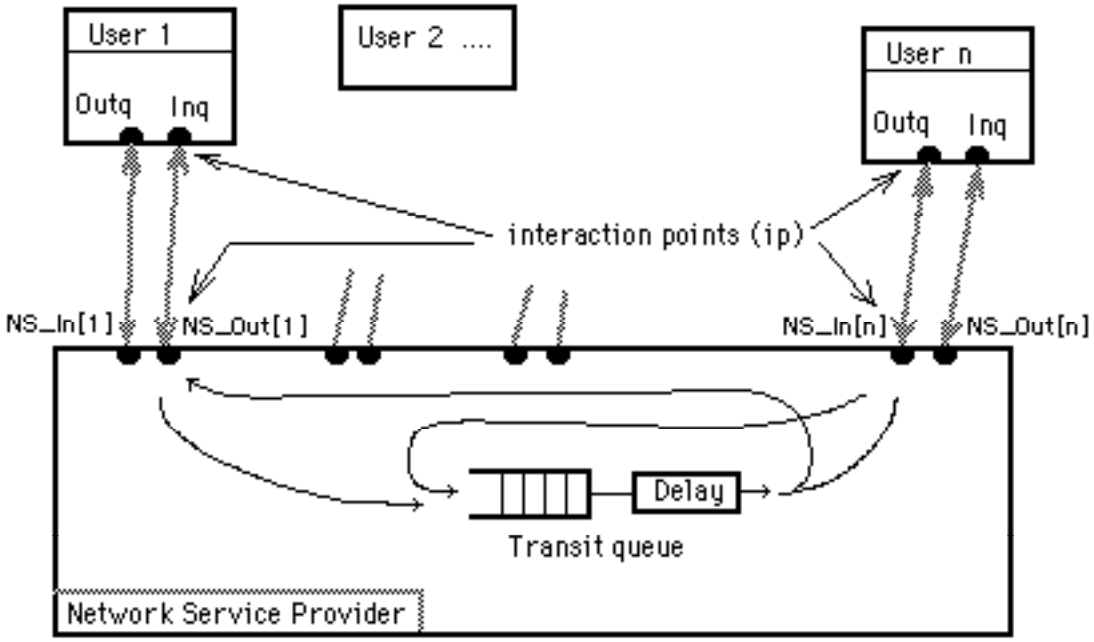

Figure 2 Simple Network

Figure 3: Simplified Network Service Specification

To test and demonstrate Simulation extensions to FDT
- Channels with delays
- Spontaneous transitions
- Resources and the Hold construct
- Delay for random times in spontaneous transitions

Definition of channel and messages

```
1.   type message_kind = (normal, retransmission,  acknowledgement);

2.   channel NSAP_primitives ( provider, user );
     by     user, provider :          MESSAGE (kind : message_kind );
```

Service provider module

```
3.  module NS_provider;
4.      ip
            NS_in,
            NS_out : array [ N_address_type ] of NSAP_primitives ( provider );
5.      internal ip
            transit_queue   fifo delay  normal(avg_transit, std_dev)
            : NSAP_primitives ( user );
6.      var
            IP_resource : array [N_address_type] of resource;
7.      initialize
            begin
                IP_resource[1] := newresource (' User-1 channel',1);
                ...etc....etc...□
            end;

8.      trans  when   NS_In [addr] . MESSAGE (kind)
               weight        1000
               hold     IP_resource[addr]  for  K1 * message length
               begin
                    output transit_queue . MESSAGE (kind)
               end;

9.      trans  when   NS_In [addr] . MESSAGE (kind)
               weight        1
               hold     IP_resource[addr]  for  K1 * message length
               begin  (* loss *) end;

10.     trans  when   transit_queue . MESSAGE (kind)
               begin
                    output     NS_Out [dest_address] . MESSAGE (kind)
               end;
    end NS_provider;
```

USER DEFINITION: a module

```
11. module user;
12.     ip      Inq, Outq : NSAP_primitives ( user );
13.     state   basic, waiting;

14.     initialize to     basic;

16.     trans   from basic      to  same
                when    Inq. MESSAGE (kind)
          begin  if kind <> acknowledgement
                        then  output Outq . MESSAGE (acknowledgement)
          end;

17.     trans   from  basic     to  waiting
                provided        true            (* spontaneous *)
                delay               uniform( a,b,U)
                begin
                        output Outq . MESSAGE (normal);
                end;

18.     trans   from  waiting   to  basic
                when            Inq. MESSAGE (kind)
                provided        kind = acknowledgement
          begin  end;

19.     trans   from  waiting   to  same
                provided        true            (* spontaneous *)
                delay           timeout
                begin
                output          Outq . MESSAGE (retransmission);
                end;
        end user ;
```

```
process user [In, Out] : noexit :=
        (   ( basic [In, Out] |> Out !normal )
        || ( i wait uniform (a,b,U); Out !normal )  )
        ; waiting [In, Out]
        where process basic [In, Out] : noexit :=
                In ? m:message_type ;
                (     [ m <> acknowledgement ] -> Out !acknowledgement
                []    [ m =  acknowledgement ] -> i )
                basic [In, Out]
        endprocess
endprocess

process waiting [In, Out] : noexit :=
          ( In !acknowledgement ;  user [In, Out] )
        [] ( i wait timeout ; Out !retransmission ; waiting [In, Out] )
endprocess
```

Figure 4: Specification of the User Process in LOTOS

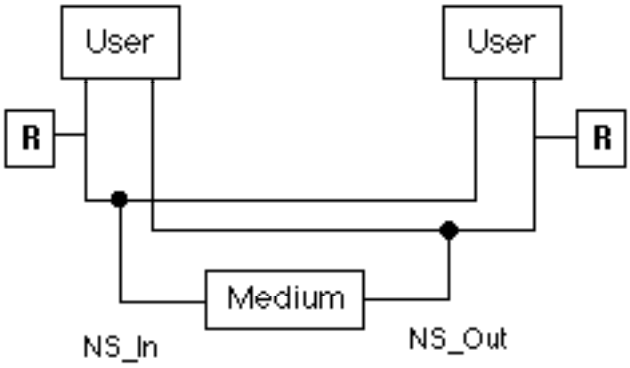

Figure 5: Lotos Model of Network System