Performance simulation of communication protocols based on formal specifications

Gregor v. Bochmann, Daniel Ouimet, and Jean Vaucher
Département d'IRO, Université de Montréal, Canada

Abstract

In most cases of system specification, the logical behavior of the system is defined separately from the performance aspects. However, this separation leads to difficulties when the performance aspects are closely related to particular logical aspects of the system behavior. This paper shows how to include the performance aspects of a system when the logical behaviour is described by means of an extended finite state machine formalism. Performance extensions to the Estelle specification language are defined and the design of a corresponding simulation system is discussed. An application of this combined approach to the OSI Transport protocol is described. The paper also includes a discussion of the main results obtained from the analysis of the logical behavior of the specification and a comparison of its predicted performance with a real implementation.

1. Formal specifications for describing logical and performance aspects

Specifications are crucial to the software development cycle. Though specifications are mainly written in natural language, the use of more formal means of expression is advantageous either to reduce ambiguity or to allow automation of certain steps in the design process.

In the area of distributed systems, an accepted formalism well suited to the expression of logical behaviour is that of finite state machines (FSMs). More recently, standards organizations have designed special languages such as Estelle [Budk 87], LOTOS [Bolo 87] and SDL [Beli 89] for the formal specification of OSI communication protocols and services. An overview of the importance of these so-called Formal Description Techniques (FDT's) and their associated tools in the areas of protocol design, implementation and testing is given in [Boch 90g] and [Lour 92].

Although these languages are well suited to express the required logical behaviour of a

system, they are not designed to specify performance aspects. For the description of these aspects, it is often sufficient to define relatively well-understood performance parameters, such as transmission delays and maximum throughput. In the case of more complex functions, however, the performance parameters are more closely related to the logical functions performed by the system. In order to evaluate the performance of a given system design, one usually refers to specific performance models, such as Markov processes or queuing networks.

Unfortunately, handling the performance aspects independently from the logical aspects of the system leads to difficulties, especially when performance is closely related to particular logical aspects. In the case of communication protocols, for instance, the error rate of the underlying communication medium (a performance aspect) together with the procedure for error detection and recovery (a logical aspect of the protocol) determine the error rate and speed of the communication service provided to the users (performance aspects).

In order to combine the logical and performance aspects of a system design within a single description, a number of specification methods combining logical and performance aspects have been proposed. Methods based on the logical aspects of finite state machines [Rudi 83b, Krit 86] or Petri nets [Moll 82, Razo 84b, Holl 87] take advantage of their simple structure, but are not powerful enough to model all aspects (logical as well as performance) of most real examples. In other cases, performance simulations have been partially based on the logical behavior of protocol specifications written in an extended finite state machine formalism [Wolf 82].

In discrete simulations, resource contention and performance are often central to the problem being studied. As a result, specialized simulation languages such as GPSS and SIMULA, embody primitives specifically designed to describe performance aspects of systems.

In this paper we consider extending the existing specification languages Estelle or SDL, which are based on an extended finite state machine formalism, with concepts derived from traditional simulation languages. The extensions cover specification of delays and duration, resource usage and stochastic choice. A more detailed discussion of this combination is given in [Boch 88b], and the application of the same approach to the specification language LOTOS is described in [Rico 91]. These FDT's are complete specification languages powerful enough to describe practically all logical aspects of the specified system.

Therefore, no simplification of the logical system aspects is necessary when the performance aspects of the system are considered.

In the software development cycle, specifications are successively refined and transformed phase by phase until executable code is obtained. Specifications are also crucial to the testing and validation process. When specifications are expressed in a formal language, some of these steps can be automated.

In support of the implementation process, translators have been designed to convert FDT protocol specifications into executable programs which implement the protocol. For the validation of the system specification, the logical and performance aspects could be analyzed through various approaches. While the logical aspects of the specification may be validated using program proof or symbolic execution techniques, these approaches become very difficult to realize in most practical situations. Therefore one of the following approaches may be more appropriate:

(1) Complete analysis of a simplified model of the logical aspects, e.g., finite state machine model, using a form of reachability analysis (see for instance [Zafi 80] or [Pehr 90]).

(2) Simulated execution of logical aspects of the specification in a kind of testing environment [Alga 93].

(3) Performance simulation, combining point (2) with the performance aspects of the specification.

During the phase concerning validation of the system implementation, the combined logical and performance specification could be used as reference against which the implementation is validated. For example:

(4) the logical aspects of the specification can be used to determine which tests should be applied to the implementation in order to find any logical errors in the implementation [Sari 89c],

(5) the specification can also be used as an oracle to determine whether, for a given test, the trace of observed interactions conforms with the specification [Boch 89m],

(6) the performance aspects of the specification provide a guideline for determining the performance aspects of the implementation to be measured, and the desired performance objectives to be attained, and

(7) the performance simulation in combination with the measurements performed on the implementation can be used to identify performance bottlenecks within the implementation.

In the following section, we start with a brief introduction to Estelle, then we present our proposed extensions for the description of performance aspects of behaviour. An Estelle description extended with performance statements can be viewed as a simulation program for the specified system. Accordingly, we have devised a simulation package which complements an Estelle compiler previously written by our group. This implements the approaches (2) and (3) to protocol analysis. The simulation tool is described in Section 2.3. To show the benefits of a simulation approach using the extended FDT, the OSI Transport protocol (class 4) was specified and simulated. This illustrates approaches (2), (6) and (7). This is described in Section 3.

2. Performance aspects for extended state transitions models

This section begins with a short introduction to the specification language Estelle [Este 89]. This language, based on an extended finite state model, was designed to express only logical aspects of behavior. We show how performance aspects can be added to Estelle and finally we present a simulation system which implements those ideas.

2.1 Specification language for extended state transition models

Specification languages such as Estelle and SDL are based on the model of finite state machines (FSM). These languages allow for extensions which deal with the parameters of interactions, additional state variables and the composition of several machines into one system. To clarify the presentation, we concentrate in the following on the FSM aspects of the languages. We consider 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: input transitions

and spontaneous transitions. 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. 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.

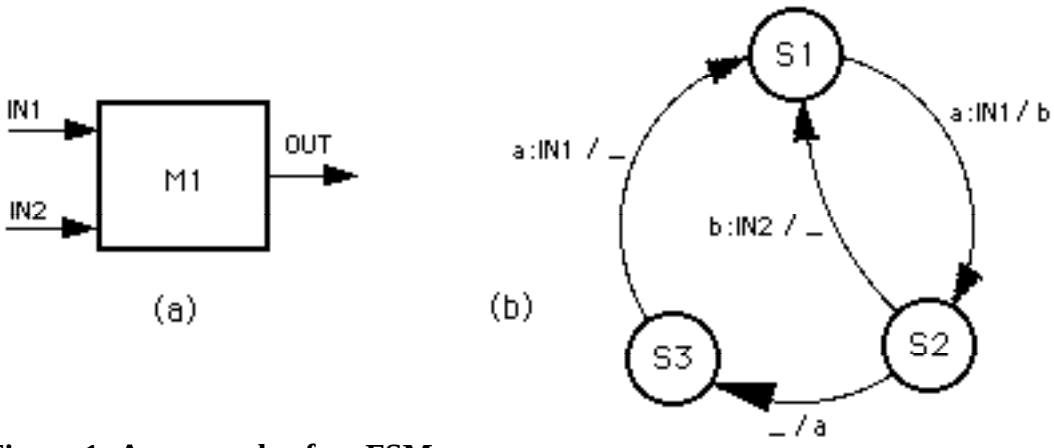

Figure 1- An example of an FSM
(a) diagram showing FSM and several I/O streams
(b) state transition diagram

We give in the following a short introduction to Estelle and then parts of the example given in the Annex. Using the syntax of Estelle, the FSM shown in Figure 1 could be written as:

```
module      M1_type;
   ip IN1, IN2, OUT ;        (* interaction points *)
end;
body        M1 for M1_type;
   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
      begin end;
   trans    from S2 to S3 (* spontaneous transition *)
      begin output OUT.a end;
   trans    from S3 to S1 when IN1.a
      begin end;
end;
```

In Estelle, a FSM is described by a **module** (declarative part) and a **body** (behaviour part).

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; spontaneous transitions have none. Finally, output statements specify the creation of messages. The notation for I/O is **<interaction_point>.<message>**.

In order to give a more meaningful example, we explain in the following the structure of the network service specification given in the Annex. For distributed systems design, the separate specification of services and protocols represent important design steps. In the case of the service specification, the distributed system is described as a single black box providing communication service to the users over physically distributed service access points (see Figure 2). The protocol specification gives a more detailed picture of the system by identifying several physically distributed "protocol entities" which provide the required service at the service access points, and the "underlying communication medium" which provides a more basic communication service through which the protocol entities exchange so-called protocol data units (PDU's). This system structure is shown in Figure 3. The protocol specification is a specification of the behavior of the protocol entities.

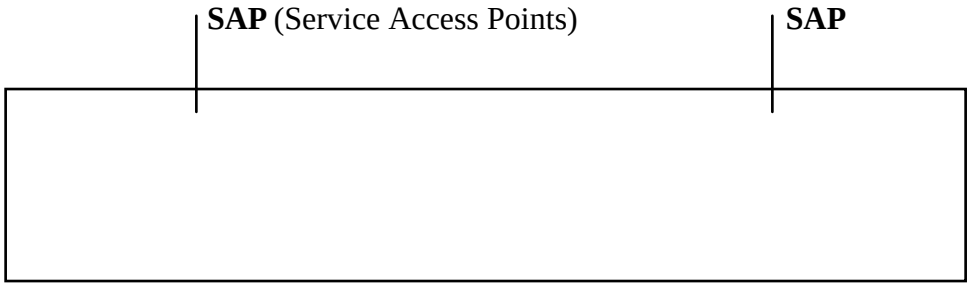

Figure 2- Service view of a distributed system

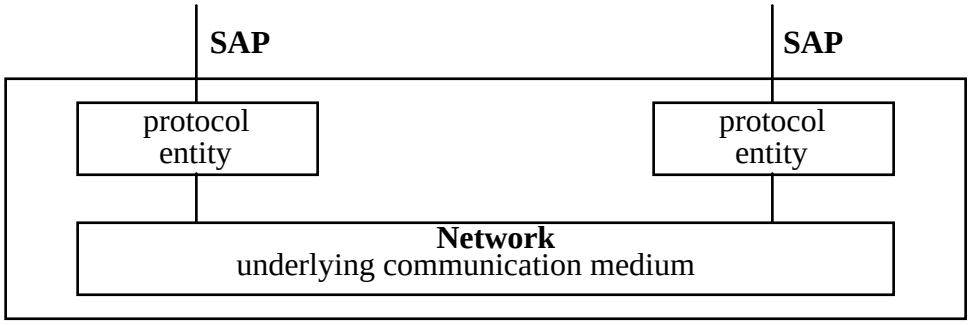

Figure 3- Protocol view of a distributed system

The structure of the *Network_service* specified in the Annex may be represented as shown in Figure 4. It is used to simulate a Network service between two Users (in our case two

Transport Protocol entities). These interactions can be executed at one of two interaction points, called *NSAP* for "Network Service Access Point", and identified by an index which takes on the values "*A*" or "*B*". This index is used by network users (higher layer protocols) to identify which side of the network is used for sending or receiving messages. A message sent on side *A* of the network should then be received later on side *B*. This is shown also in the specification of the network in the Annex when the function *opposite* changes *A* in *B* and *B* in *A*.

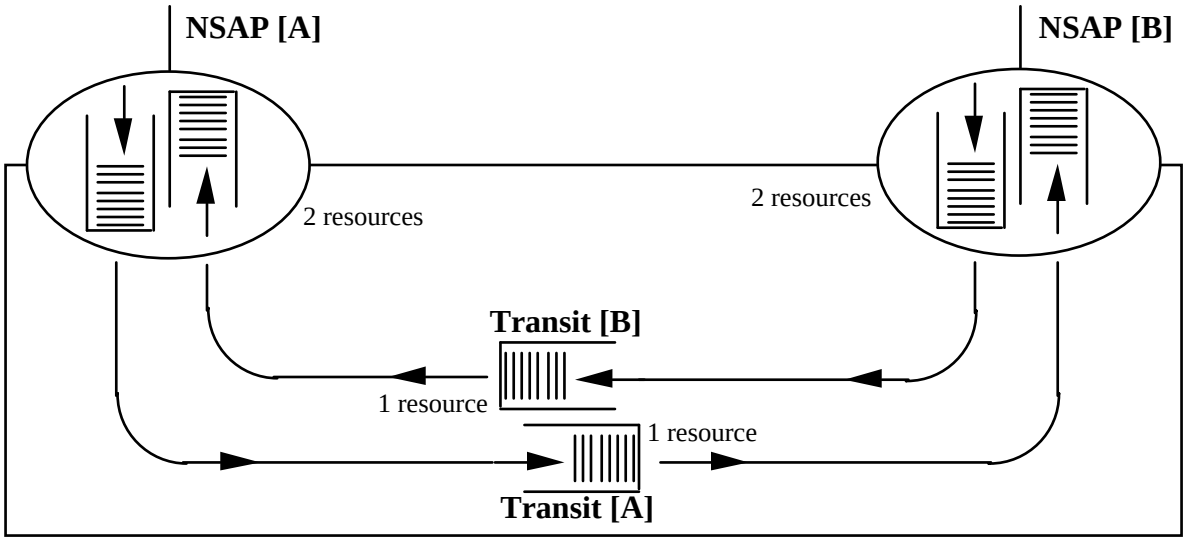

Figure 4- Structure of the Network module specified in the Annex

In the following, line numbers relate to the lines of the Annex. The *Network_service* module defines four states *Idle*, *Opening*, *Open* and *Close* (line 51) and four transitions (lines 72, 81, 95 and 103). The machine is *Idle* at the beginning. When it receives a *N_CONNECT_req*, it goes to the *Opening* state and sends a *N_CONNECT_ind* to the other side (lines 95-101). It then waits until reception of *N_CONNECT_req* when it goes into *Open* state and sends the primitive *N_CONNECT_ind* to the other side (lines 103-109). Disconnection is done similarly but not shown in the Annex. Reception and transmission of data are handled in two transitions (lines 72-90) while the finite state machine is in the *Open* state:

```
72      TRANS
73      any side: Side_type do
74      when NSAP[ side ].N_DATA_req (* msg *)
75      from Open to Same
76      hold Incoming_NSAP_resource[ side ] for Interface_hold
77          begin
```

```
78          output Transit[ side ].N_DATA_req(Msg);
79          end;
80
81      TRANS
82          any side: Side_type do
83          when Transit[ side ].N_DATA_req (* msg *)
84          from Open to Same
85          hold Propagation_resource [side] for Propagation_hold
86      begin
87      output NSAP[ Opposite(side) ].N_DATA_req(Msg);
88      (* There will be a Hold on the resource Outgoing_NSAP_resource[ side ]
89       for Interface_hold when this signal arrives at the Network User. *)
90      end;
```

The necessity of two transitions is explained later. One transition simply takes data from the *NSAP* interaction point and send it to an internal interaction queue *Transit* (line 78), and another takes the data from the same queue and send it to the other *NSAP* (line 87). Statements of lines 76 and 85 are performance extensions to the language and will be explained in Section 2.2. The *any* statement (lines 73 and 82) enables the transition to be fired with any one value given by the type *Side_type* (*A* or *B* in this case), and then executes the transition body part with the *side* variable instantiated to this one value.

In order to describe more complex systems, several state transition modules can be interconnected such that the output from one module becomes the input to another one. An example is shown in Figure 5, which shows two *Transport Protocol* modules communicating through a *Network_service* module. In Estelle and SDL, the output of one module first enters a queue before it is processed by the destination module. In SDL, each module has exactly one such queue through which all inputs must pass; Estelle allows one input queue per interaction point and this is what we assume in the following. For instance, the *Network_service* module has two input queues (line 29), one for each user module.

```
29      NSAP:    array [ Side_type ] of NS_primitives( N_provider )
30          individual queue
31          fifo delay Interface_delay;
```

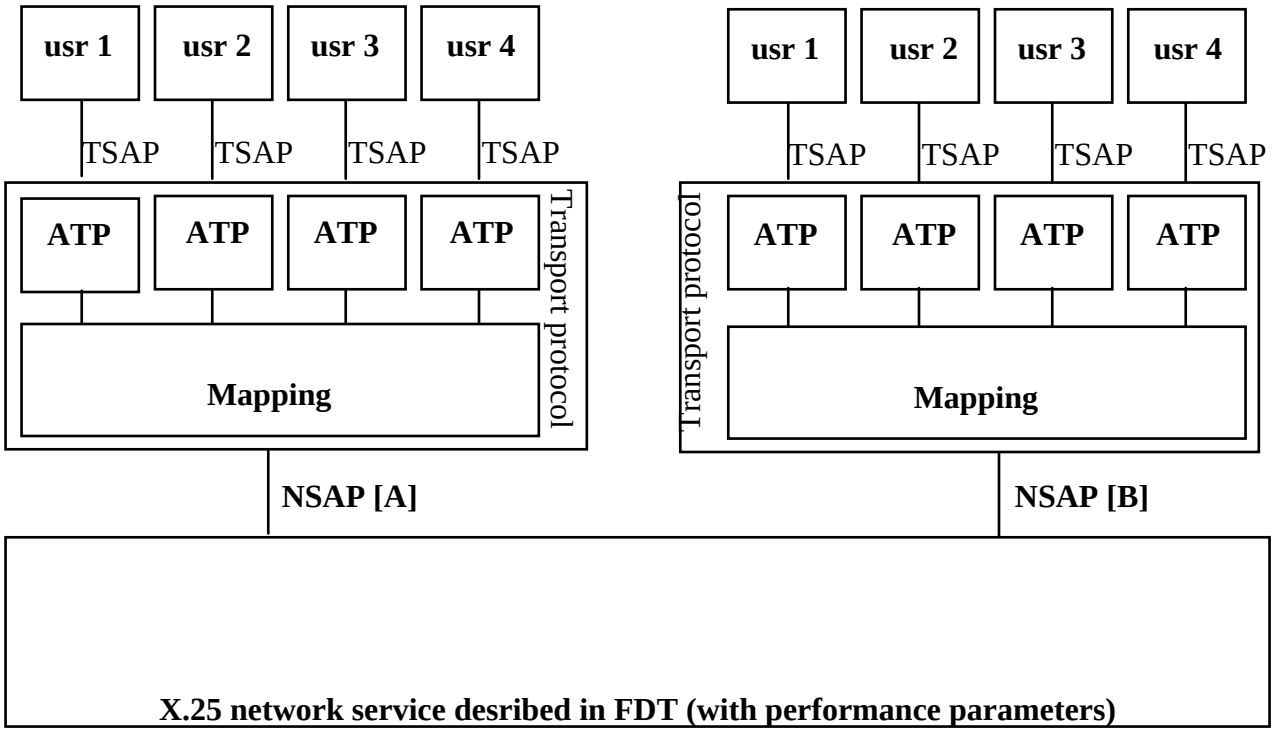

Figure 5- FDT simulation module with Transport protocol

2.2 Language aspects for performance description

Different approaches can be taken to define performance aspects of state transition models. In a basic Markov model, a transition is instantaneous, but some time elapses between the instant when it could occur and when it does occur. For each transition t of the machine, a distribution function *pt(T)* defines the probability that the machine does this transition within T time units after the transition has become possible, assuming that no other transition has been executed.

In a more complex performance model, based on transition times, 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 *St(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 model shared resources more naturally with FIFO queuing of requests. The requests wait in the input stream until they are processed, and the processing is modeled by the transition and its execution time. Only one request (transition) is processed at a time by any given FSM. Various variations of these basic models have been described in the literature; for example, [Moll 82] and [Krit 86] belong to the Markov and transition time performance models, respectively.

Another discipline concerned with the specification of the performance aspects of parallel systems is simulation. The knowledgeable reader will note the influence of languages such as GPSS and Simula in the performance extensions to the finite state model that are described in the following sections.

2.2.1. Resources

In our system, a transition may take some time. The duration of a transition is expressed by a **HOLD** clause (extension of Estelle) of the form

hold for <expression>

where <expression> is a real value expression in time units. The module within which a transition executes is considered to be a resource required for the duration of a transition. Thus only one transition at a time may take place in a given module. During the execution of a transition in a module, no other transition may be performed by the same module. The outputs generated by the transition are available at the end of the transition.

For certain applications, it was found convenient to introduce other 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 the Annex receives input packets over two interaction points. In order to model the maximum throughput available for a given interaction point, a corresponding resource *Incoming_NSAP_resource* is declared (line 49) and the input transitions receiving a packet hold the corresponding resource for the time proportional to the length of the packet received (lines 76).

```
49      Incoming_NSAP_resource: array [ Side_type ] of resource;
...
76      hold Incoming_NSAP_resource[ side ] for Interface_hold
```

If a 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>.

2.2.2. Transition probabilities

In the underlying finite state transition model, there may be several different transitions enabled for a given state and input signal. Like in the Markov model, one may associate different probabilities with these transitions. However, in the context of an extended state transition language such as Estelle, the situation is not that simple. Due to the extensions involving interaction parameters and additional enabling conditions for the transitions, which usually depend on parameters and additional state variables, the present major STATE and the available kind of input interaction do not completely determine which transitions are possible. Therefore it is not clear how transition probabilities (which must add up to one) 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, we have adopted a notation where the 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 case of SDL, the situation is somehow different. There is 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 of CASE statements may be non-deterministic. It is therefore sufficient to provide random distribution functions (see below).

2.2.3. Interaction queues with transmission delays

For modeling 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 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 modeled by associating a resource with the input to the medium (Section 2.2.1). Its service time will limit the number of transmission requests that can be handled. Transmission delays are described by introducing additional properties for the input queue associated with a given interaction point. The syntax of the interaction point declaration becomes

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

where <properties> may 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 were 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 shown in the Annex, transmission delays are modeled by using a FIFO queue (*Transit*, see also Figure 4). 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 and has random delays with a normal distribution (line 43).

43 **fifo delay** Propagation_delay;

A separate transition (line 81, see above) removes the messages from the *Transit* queue and places them on the output interaction point.

2.2.4. The DELAY clause

The DELAY clause for spontaneous transitions is already defined in Estelle. In a more restricted form, it can be written as:

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 units (if transitions are executed during this time interval, the condition must remain true in between the transition executions). This clause can be used to describe timeout transitions.

2.2.5. 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 in functions such as *Interface_hold* (line 22) or *Interface_delay* (line 20).

```
20      function Interface_delay: integer;      primitive;
21      function Propagation_delay: integer;    primitive;
22      function Interface_hold: integer;       primitive;
23      function Propagation_hold: integer;     primitive;
```

2.3 A Simulation System

The performance aspects described previously were tested in a simulation tool for Estelle specifications. The approach chosen was to combine an Estelle compiler and execution package with a process-oriented simulation package handling the time aspects.

The specification language that we have used for the simulation system is an early version [FDT 83] of Estelle [Este 89], which differs from the latter by a static module structure and some small syntactic changes. For our purposes, the restriction to a static module structure presented no problem. The Estelle compiler [Boch 87h] accepts as input a system specification written in Estelle and produces as output Pascal procedures. For each module type in the specification, the compiler generates one procedure. A run-time execution package, associated with the compiler, uses these procedures for the execution of the specified system. Originally, the compiler and the associated run-time system were designed to provide an implementation-oriented run-time environment [Boch 87h, Vuon 88]. In fact, the compiler and the run-time package have been used for a number of protocol implementation projects [Boch 86n], [Boch 87h] and [Boch 89m].

The run-time package keeps track of all module instances in the specified system during the execution phase, and of all output interactions which are placed in the appropriate input queues. Based on some implementation-dependent scheduling scheme, it selects an input transition or a spontaneous transition of a particular module instance for execution. It then calls the generated procedure corresponding to the module type, and provides the input interaction (if any) and the state information of the module instance as parameters to the procedure. The generated procedure does the processing corresponding to the transition and calls an OUTPUT procedure for each output interaction generated. The latter procedure is part of the run-time package.

In order to introduce the performance aspects in the run-time environment, it was necessary to modify the run-time package associated with the Estelle compiler and combine it with a time simulation function. An existing process-oriented simulation package PSIM [Vauc 84b] was adapted for this purpose. The list handling routines, the tracing mechanism, the dynamic allocation, disposal routines and the random number generators could be used without modification. However, the event notice structures, all scheduling routines, the simulation executive and the resource algorithms had to be redesigned for the Estelle environment. This was made necessary both by the requirement to interface with the fixed structure of the automatically generated procedures (the Estelle compiler was not changed), as well as by the very specific semantics of some of the interaction primitives: for example, spontaneous transitions with both DELAY and HOLD clauses. The resulting simulation package is further described in [Vauc 84].

This combination of an Estelle compiler and a simulation package provides a tool which allows the automatic execution of simulation runs of specification systems, as described in Section 3, except for one point. The performance clauses in the specifications, written in the notation described in Section 2.2, must be hand-translated into corresponding Pascal statements and/or declarations in the generated procedures. The translation scheme is quite simple and further described in [Vauc 84].

3. Simulation studies of the OSI Transport protocols

A relatively extensive simulation study of the OSI class 4 Transport protocol [OSI TP], in the following called TP-4, was performed with the objective of examining the utility of the simulation approach described in this paper (direct execution of a formal specification). The main characteristics of this experiment and the major results are described in this section. Further details can be found in [Boch 86m].

3.1. Overview

The overall structure of the simulated system is shown in Figure 5. Two Transport entities execute the TP-4 protocol by exchanging PDU's over a simulated Network connection and provide the Transport communication service to four users each. Each user module can

establish a Transport connection with a user connected to the other Transport entity. Up to four Transport connections can be simulated simultaneously. The simulation study concentrated on the data transfer phase.

The Network specification used for simulation was similar to the specification shown in the Annex (only one additional internal queue representing VMS-mailboxes has been removed to simplify the code presented in this paper). The specification of the Transport entities was adapted from a preliminary version of a formal Transport protocol specification developed within ISO [NBS 85]. That specification had been adapted for the development of a semi-automated implementation and was subsequently further adapted in order to obtain a "parameterized" specification [Serr 86b], which has a number of parameters, the values of which determine selected implementation options. Originally developed for the purpose of testing [Sari 86b], the parameterized specification is therefore also very suitable for simulation studies since different implementation options can be simulated.

The simulation study proceeded through the following phases:

Phase 1: Network simulation only,
Phase 2: Basic Transport simulation (no processing overheads),
Phase 3: Transport simulation with realistic CPU and buffer constraints,
Phase 4: Transport simulation for the case of packet losses.

In order to have a point of reference from which the performance results of the simulation could be validated, it was decided to try to reproduce, through simulation, the behavior of the real implementation. The implementation was also based on the "parameterized" specification [Serr 86b] and was running under the VMS operating system on a DEC VAX computer, using the Network service provided by the DEC X.25 software over the Canadian public packet-switched data network. The measurements were taken in a configuration shown in Figure 6 where all users were on the same computer, and the network connection looped back to the computer from the network node to which the latter was connected.

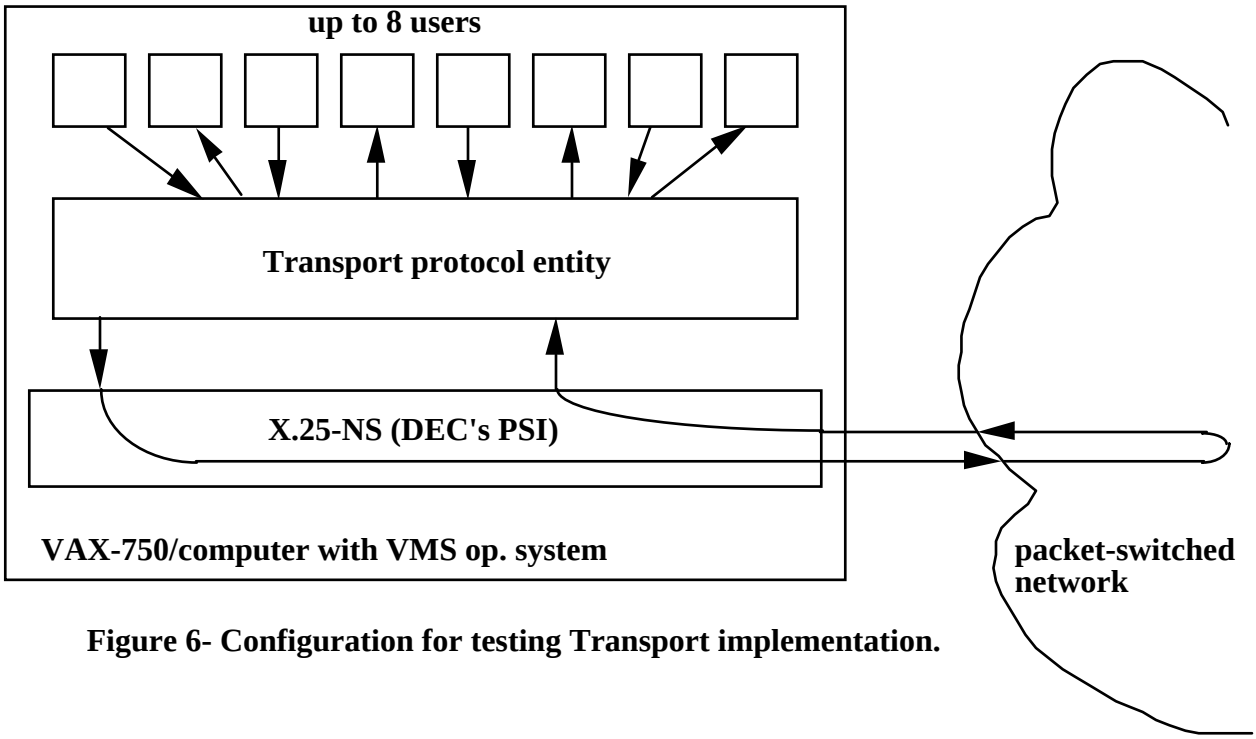

Figure 6- Configuration for testing Transport implementation.

During phase (1) the performance parameters of the Network simulation were adjusted to the Network performance measurements made. Phase (2) was introduced to validate the logical correctness of the simulated formal specification of the Transport protocol. Phase (3) was used to adjust certain PDU utilization and buffer management parameters such that the measured Transport performance would be reproduced. This was possible by introducing a single HOLD clause in all transitions of the Mapping module. Phase (4) gave some results for expected performance if packets were lost occasionally. These results were not compared with experiments on a real implementation, since the real network was reliable. Several simulation results indicated quite bad performance for the protocol. These results could be used to find ways to introduce essential performance improvements for the implementation.

3.2 Selecting the simulation parameters

In order to study the performance aspects of the Transport protocol in a systematic manner, the following traffic patterns were selected. They were used for the simulations, as well as for measurements on a real implementation [Serr 86b].

(a) a single connection with interactive traffic,

(b) four multiplexed connections with interactive traffic,

(c) a single connection with file transfer traffic,
(d) four multiplexed connections with file transfer traffic, and
(e) three connections with interactive traffic and one connection with file transfer traffic.

In all cases, a single Network connection was used. For interactive traffic, an exponential data departure rate was assumed, and the data block size was exponentially distributed with an average length of about 8 octets. For file transfer traffic, the data departure rate was constant, and the data block size was fixed to 123 octets (which can be sent in a single X.25 data packet). A typical simulation measurement involved three simulation runs (with different random number seeds) of 500 to 1000 data blocks sent. In order to allow the system to enter a stable situation, the first 64 data blocks were not used for the measurements. The traffic is generated by the user modules, which are specified in Estelle using spontaneous transitions with DELAY clauses and random distribution functions.

An important aspect of any simulation system is the collection of results. Our simulation system provided accounting for the occupation of each resource, however, measurements for delay and interarrival rates had to be explicitly programmed by including statements in the user modules for writing the relevant information on a trace file. In order to calculate delays, each data block included a time-stamp containing the time when the block was sent by the user module. In addition, a sequence number was included which was used to check the correctness of the error recovery mechanism of the protocol.

3.3 Selecting realistic performance parameters

The simulation performance parameters were adjusted to reproduce the characteristics of the real Transport implementation shown in Figure 6. In addition to the measurements of delay and interdeparture and interarrival rate, the measurements on the implementation also included the determination of CPU usage by the Transport protocol task.

In Phase (1), the Network parameters were adjusted. Since our implementation used a relatively slow network access line (2400 bits per second), a flow control window of 2, and the loop-back network connection shown in Figure 6, it could be assumed that the Propagation_delay and Propagation_hold values (see lines 20-23 of the Annex) are negligible compared to the corresponding values at the network access interface. The

Interface_delay value was determined by measuring the transmission delay of data blocks exchanged between user tasks directly connected to the Network service (without the intervention of the Transport protocol task). The Interface_hold value was determined by measuring the maximum throughput obtainable by the implementation. The following values (in milli-seconds) were obtained, assuming a linear dependency on the size of the data blocks:

Interface_delay = 120 + n * 5.2

Interface_hold = 60 + n * 3.6

where n is the number of octets in the data block. The hold value corresponds to a little over 80% utilization of the network access line, which corresponds to 100% if one considers the transmission overhead of the X.25 packet and link layers.

These obtained network performance parameters were used in all the subsequent phases of the simulation study. The processing delay of the Transport entity was assumed to be zero during Phase 2, while during Phase 3, an attempt was made to adjust the performance parameters of the Transport entity in the simulation to match the measured performance of the implementation.

In order to simplify the performance simulation, we assumed that the processing delay of the Transport entity is located in a few transitions only. As shown in Figure 5, each entity contains a Mapping module which looks after the multiplexing and coding of the PDU's, and one ATP module for each active Transport connection (for more details, see for example [Boch 90a]). We assumed for the simulation that all transitions of the Mapping module involve the same processing delay (independently of the size of the data block, since data buffers are referenced, not copied), and that the transitions of the ATP modules have no processing delay. With this assumption, the measurements lead to a processing time of 175 milli-seconds for each Map transition. However, there is an additional processing delay which is due to the fact that the CPU of the computer is time-shared with other operating system tasks. Since the operating systems used a CPU quota of 200 milli-seconds (maximum time a task can keep the CPU), we obtain a Hold time for the Map transitions equal to

Map_hold = 175 + DRAW (0.6*(175/200)) * 200

We assume here that the Transport entity has always input to process (see Section 3.5.2) and that 175/200 of the time the processing of an input extends to the next time slot. The CPU utilization by other processes has been measured to be 60%; this implies that (175/200)*60% of the time the Transport entity has to wait for another time slot because

the CPU is seized by another process.

3.4 Typical simulation results

Simulations were executed for each of the five cases mentioned in Section 3.2. About three to ten simulations for each case and set of parameters were sufficient to get reproducible results. They were long simulation executions of many thousands of messages (usually 1000 messages sent for each user). In order to obtain results for stationary system states, the first hundred of messages in each simulation were not accounted for. Graphs were made showing the relation between the number of message sent per second and their end-to-end delay, and the resources usage.

Initially, as the rate at which messages are submitted is increased, the message throughput (departure rate) increases. However, there is an upper limit to throughput (around 1 message/second in Figure 7). Figure 8 shows the CPU to be the bottleneck: CPU utilization is proportional to the throughput and reaches 100% for the observed maximum throughput. As could be expected from queuing theory, the transmission delays increase exponentially as the utilization of the CPU reaches 100% (see Figure 7). Figures 9 and 10 for the case of four users are quite different than the ones for a single user. There is still a maximum throughput around 1/4 message/sec/user (or 1 message/user overall), but attempts to increase throughput by increasing submission rates only results in lower throughput (around 0.6 messages/sec).

Further study showed this phenomenon to be the result of the flow control mechanism. In our case, each user was allowed 16 credits: that is he could have a maximum of 16 unsent messages waiting in the system queues. After that he would be allowed to provide a message only when one had effectively been sent. When a system is saturated, the delay through the system is directly proportional to the queue length. In our case, with one user we saturate with 16 messages in the system but with four users, there are 64 messages and delays are four times greater for submitted messages. This difference becomes significant because with longer delays, retransmissions due to time-out were activated, thus increasing congestion even further. Note that in all cases the CPU proved to be the bottleneck with 100% utilization at saturation. The drop in throughput at saturation for the four users case from 1 message/sec to 0.6 message/sec would suggest that 40% of all transmissions were

duplicated messages.

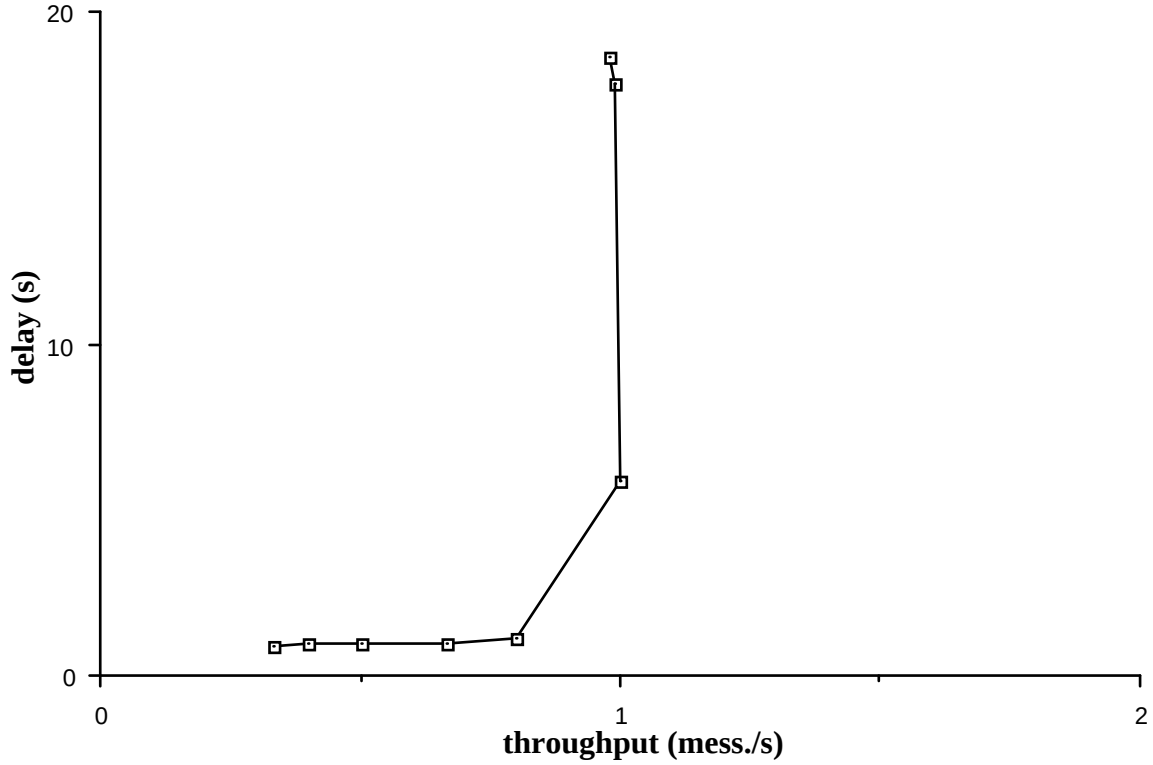

Figure 7- TP4 with one user and short messages: throughput vs end-to-end delay

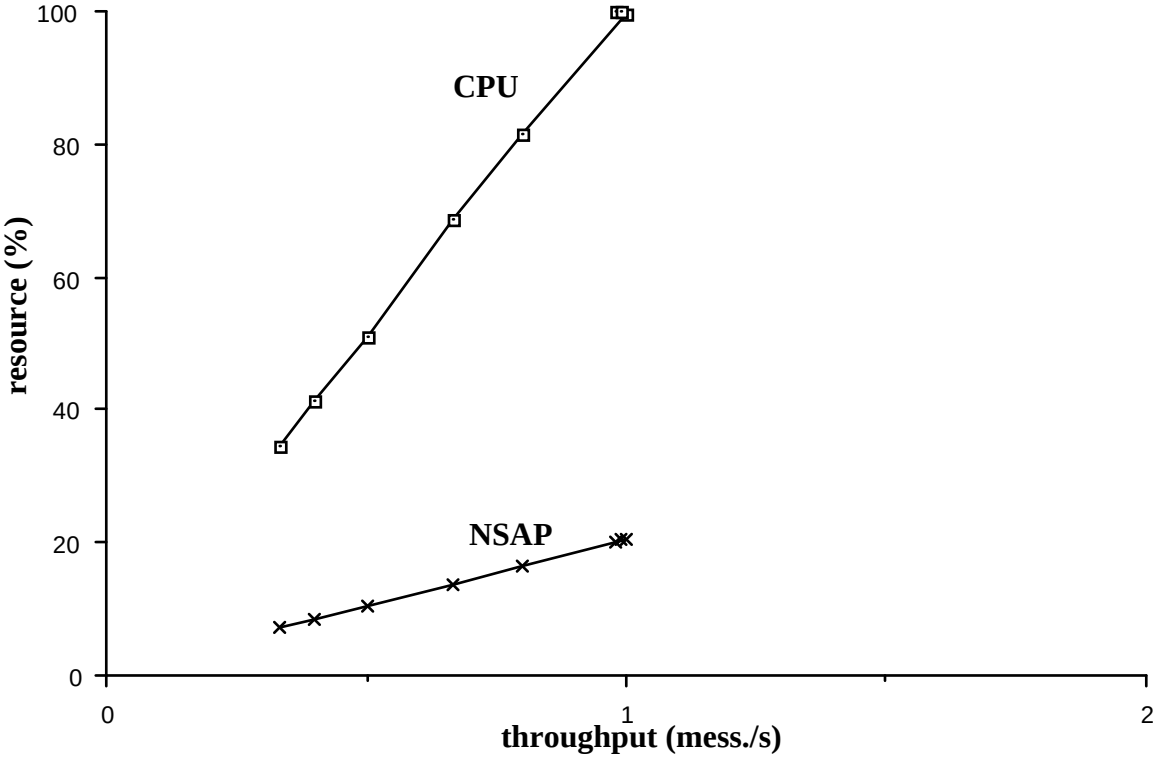

Figure 8- TP4 with one user and short messages: throughput vs resource utilization

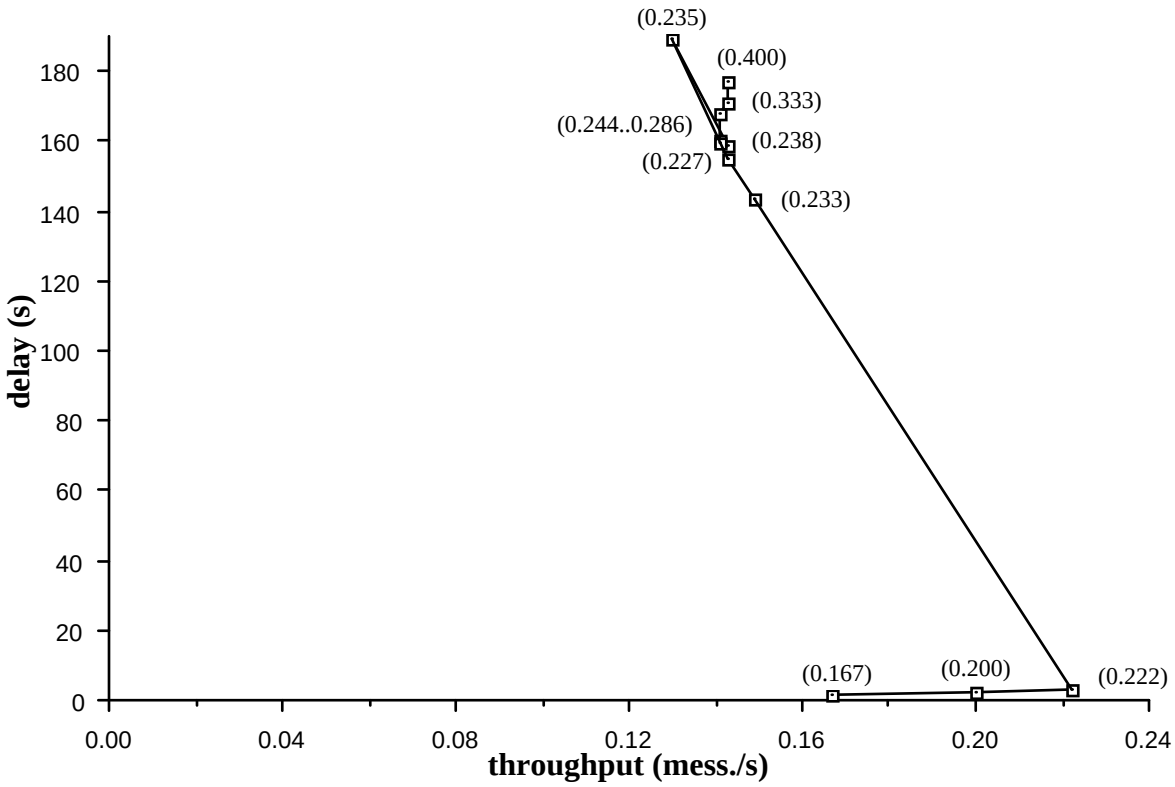

Figure 9- TP4 with four users and short messages: throughput vs end-to-end delay

Note: the figures in parentheses indicate the nominal desired throughput (the actual throughput is limited by flow control

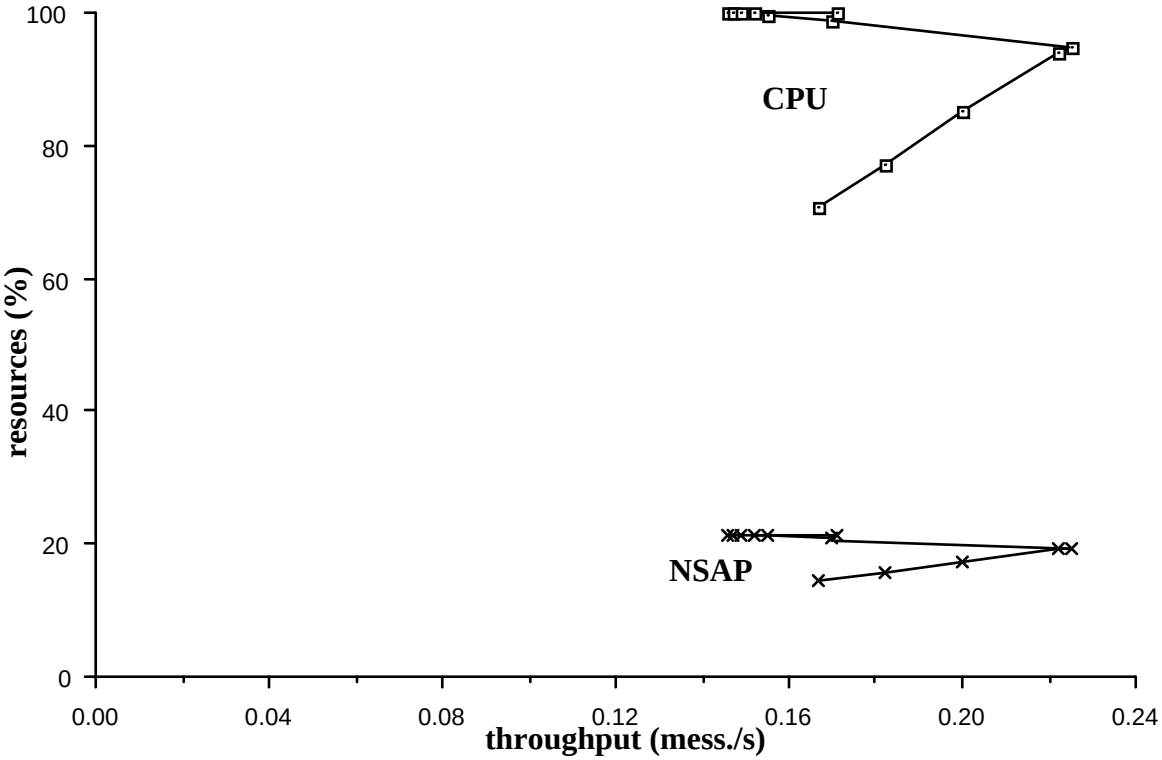

Figure 10- TP4 with four users and short messages: throughput vs resource utilization

3.5 Lessons learned

From our experiments, we noted two areas for which the simulation studies were particularly useful.

3.5.1. Testing the logical aspects of the formal specification

A formal specification of a protocol (or any other system of interest) must be validated. In the case of executable specifications, one approach to validation is the execution of test scenarios. The semi-automatic implementation approach described in [Boch 87h] can be used to obtain executable programs directly based on the formal specifications. In fact, the Transport implementation mentioned above was obtained in such a manner. It was partly debugged through the execution of a certain number of test executions. A similar, but more flexible, debugging environment is provided by the simulation system.

During Phase 2 of the simulation study, several errors of the formal Transport specification were detected. In fact, the specification had not been completely debugged during the previous implementation experiments, especially not for the case of multiplexing. Two

kinds of logical errors were detected: specification errors and implementation errors. The debugger of the VMS operating system was used to debug the TP-4 within the simulation program. After adapting certain key functions of the debugger for the FDT program, it turned out that the so created environment was very useful for debugging the Transport specification.

Our experience shows that the simulation environment is more suitable for debugging a specification than a standard testing environment. In particular, a simulation environment with simulated performance parameters allows these parameters to be changed in such a manner that execution scenarios are obtained which would usually not occur in a real implementation. For instance, we tested the behavior of the Transport entity in the presence of certain time-outs which would seldom (or never) happen in our real implementation.

3.5.2. Understanding the performance issues of the implementation

As could be expected, the simulation studies underlined certain problem areas in the implementation of the Transport protocol. Again, the ease of changing the parameters in the specification was a key factor for the success of the simulation approach. The identified problem areas included:

(1) Retransmission time-out and/or credits: The simulated system (and the implementation) allows for very large buffers between the user and the Transport entity, as well as between the Transport entity and the Network service. With 4 parallel Transport connections and 16 credits per connection, it was therefore possible to send 64 data PDU's at once into the Network layer. Because of the slow speed of the network access line, a certain number of these PDU's did not arrive at the remote site before the retransmission timer expired, thus generating more data PDU's to be transmitted. As mentioned previously, when the system is saturated, the transit time is directly proportional to the queue length. Adding credits means that more messages are stored in internal queues increasing delays to the point where time-outs occur (compounding the problem). This can only be prevented by adjusting the retransmission time-out, or the number of credits, to the transmission speed of the Network connection and the number of simultaneous Transport connections. This potential problem was identified during Phase (2) of the simulation.

(2) CPU bottleneck: As Figures 8 and 10 show, the CPU utilization reaches 100% when the traffic increases (difference between the two curves is explained in Section 3.4). This was a surprise to us, since we expected that the network access line would be the bottleneck. This lead us to consider performance improvements in the Transport implementation, in particular the reduction of spontaneous transitions to be checked after each executed transition, and the introduction of priorities for transitions, which should lead to checking the more probable transitions first.

(3) Improving error recovery: During Phase (4) of the simulation, the Transport protocol was tested for the case that PDU's are lost in the Network. This situation never occurred in our real implementation. As a result, a few specification errors and implementation choices were rectified.

4. Conclusions

The advantages of protocol simulation before implementation (or after) are numerous. First the use of one specification in FDT (the same as for the real implementation) allows us to make simultaneous corrections or modifications to both the real implementation and the simulated one. Compared with traditional queuing model or other analytical models, the simulation approach described here allows for a more representative and realistic modeling of the systems under investigation, and therefore provides a more accurate view of what really happens. An added bonus is the simulation environment which can also be used for easy debugging which should lead to the situation where most of the implementation errors can be corrected before trying the system in the real world. Also some adjustments and improvements to enhance performance can be made more easily.

Our experiments with the TP-4 implementation showed the CPU to be the main bottleneck, and other areas for improvement were also found. After these tests, results were made available to the designer of the TP-4 implementation (Jean-Marc Serre). He could easily point out the deficiencies that follow. First, credit allocation could be more intelligent and depends on the number of users and network connections (and not be constant). Some transitions in the finite state machine should have some priority and be checked first: this could improve a lot CPU time processing. And flow control between Transport Protocol and adjacent layer could be much more efficient by reducing credit given to a user of Transport depending on how many users or what kind of users they are. Then timer

adjustment could be made and improve results significantly.

At the end, some simulations were done with a Transport losing blocks over Network service. After debugging the Transport implementation, it proved to be quite reliable: no data was lost, but throughput was reduced.

Thus, we conclude that simulation should be an integrated part of the testing of any protocol. In our case (Transport class 4), it proved to be essential for the correctness and efficiency of the implementation.

Acknowledgments.
We would like to thank Jean-Marc Serre for many useful discussions concerning the Transport implementation and simulations described in this paper. This work was partly funded by the Department of Communication under contract OST85-00257 and by the Natural Sciences and Engineering Research Council of Canada.

References:

[Alga 93] B. Algayres, L. Doldi, H. Garavel, Y. Lejeune and C. Rodriguez, "VESAR: A pragmatic approach to formal specification and verification", Computer Networks and ISDN Systems, special issue on Tools for FDTs, Vol.25, no.7, February 1993, North-Holland Publ.

[Beli 89] F. Belina and D. Hogrefe, The CCITT-Specification and Description Language SDL, Computer Networks and ISDN Systems, Vol. 16, pp.311-341, 1989.

[Boch 86m] G.v.Bochmann and C.Tropper, "Evaluation of Transport protocols", Final report, prepared for CERBO Informatique Inc. under contract for DOC Canada, 200 pages, Oct. 1986.

[Boch 86n] G.v.Bochmann, "Semi-automatic implementation of Transport and Session protocols", Computer Standards and Interfaces, Vol. 5, no. 4, 1986, pp. 343-349.

[Boch 87h] G.v.Bochmann, G.Gerber, and J.M.Serre, "Semi-automatic implementation of communication protocols", IEEE Tr. on SE, Vol. SE-13, No. 9, September 1987, pp. 989-1000 (reprinted in "Automatic Implementation and Conformance

Testing of OSI Protocols", IEEE, edited by D.P.Sidhu, 1989).

[Boch 88b] G.v.Bochmann and J.Vaucher, "Adding performance aspects to specification languages", IFIP Symposium on Protocol Specification, Testing and Verification, Atlantic City, June 1988, pp. 19-31.

[Boch 89m] G.v.Bochmann, R.Dssouli and J.R.Zhao, "Trace analysis for conformance and arbitration testing", IEEE Tr. on Softw. Eng., Nov. 1989, pp.1347-1356.

[Boch 90a] G. v. Bochmann, "Specifications of a simplified Transport protocol using different formal description techniques", Computer Networks and ISDN Systems, Vol. 18,5 (June 1990), pp. 335-377.

[Boch 90g] G. v. Bochmann, Protocol specification for OSI, Computer Networks and ISDN Systems 18 (April 1990), pp.167-184.

[Bolo 87] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language Lotos", Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25-59, 1987.

[Budk 87] S. Budkowski and P. Dembinski, "An introduction to Estelle: a specification language for distributed systems", Computer Networks and ISDN Systems, vol. 14, no. 1, pp.3-23, 1987.

[Este 89] ISO IS9074 (1989) "Estelle: A formal description technique based on an extended state transition model".

[FDT 83] ISO FDT Subgroup B, "A FDT based on an extended state transition model", ISO TC97/SC16 N1347 (revised July 1983).

[Holl 87] Mark A. Holliday, Mary K. Vernon, "A generalized timed Petri net model for performance analysis", IEEE Trans. on Software Engineering, vol. SE-13, no. 12, December 1987, pp. 1297-1310.

[Krit 86] Pieter S. Kritzinger, "A performance model of the OSI communication architecture", IEEE Trans. on Communications, Vol. COM-34, No. 6, June 1986, pp.554-563.

[Lour 92] A. F. Loureiro, S. T. Chanson and S. T. Vuong, "FDT tools for protocol development", Proc. Fifth International Conference on Formal Description

Techniques (FORTE'92, Tutorials), Lannion, France, October 1992, pp.38-78.

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

[NBS 85] National Bureau of Standards, "Formal Description of the ISO 8073 Transport Protocol", May 1985.

[OSI TP] ISO TC97/SC6, IS 8073, "OSI - Connection Oriented Transport Protocol Specification".

[Pehr 90] B. Pehrson, "Protocol verification for OSI", Computer Networks and ISDN Systems 18 (1989/90) 185-201.

[Razo 84b] R.R.Razouk and C.V.Phelps, "Performance analysis using timed Petri nets", in Proceedings IFIP Workshop on "Protocol Specification, Testing and Verification", Sky Top, June 1984, Y.Yemini (ed.), North Holland Publ. Comp.

[Rico 91] N. Rico and G. v. Bochmann, "Performance description and analysis for distributed systems using a variant of LOTOS", Proc. IFIP Symposium on Protocol Specification, Testing and Verification, Stockholm, June 1991.

[Rudi 83b] H.Rudin, "From formal protocol specification towards performance prediction", in Protocol Specification, Testing and Verification, H.Rudin and C.West (eds.), North Holland Publ. Comp., 1983.

[Sari 86b] B.Sarikaya, G.v.Bochmann, M.Maksud, J.M.Serre, "Formal specification based conformance testing", Proc. ACM SIGCOMM Symposium, Aug. 1986, pp. 236-240.

[Sari 89c] B.Sarikaya, "Conformance Testing: Architectures and Test Sequences", Computer Networks and ISDN Systems 17 (1989), pp. 111-126.

[Serr 86b] J.M.Serre, G.v.Bochmann, M.Maksud, B.Sarikaya, "A parameterized specification of the OSI Transport protocol class 4", University of Montreal, prepared under contract for DOC (4ER.36100-5-0149), April 1986.

[Vauc 84] J. Vaucher and G.v. Bochmann, "A simulation tool for formal specifications" (27pages + annexes), prepared for CERBO Informatique Inc. under contract for

the Department of Communications Canada, 1984.

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

[Vuon 88] S.T.Vuong, A.C.Lau and R.I.Chan, "Semiautomatic Implementation of Protocol using an Estelle-C Compiler", IEEE Transaction on Software Engineering, vol. 13, no. 3, pp. 384-393, March 1988.

[Wolf 82] B.Wolfinger and O.Drobnik, "Simulation of protocol layers of communication in computer networks", in Computer Networks and Simulation II, S.Schoemaker (ed.), North Holland Publ. Comp., 1982.

[Zafi 80] P.Zafiropulo, C.H.West, H.Rudin, D.D.Cowan, and D.Brand, "Towards analyzing and synthesizing protocols", IEEE Tr. Comm. COM-28, 4 (April 1980), pp. 651-660.

Annex: Network Service specification

```
 1
 2    type
 3       Side_type = (A, B); (* left and right side on figure 4 *)
 4       Msg_body  = ...;
 5
 6    channel NS_primitives( N_user, N_provider ) ;
 7    by N_user :
 8          N_CONNECT_req;
 9          N_ACCEPT_req;
10          N_DATA_req(   Msg    :      Msg_body ) ;
11
12    by N_provider :
13          N_CONNECT_ind;
14          N_ACCEPT_ind;
15          N_DATA_ind(   Msg    :      Msg_body );
16    end NS_primitives ;
17
18    (* Declaration part of primitives used to evaluate queue delays and
19       hold times on resource *)
20    function Interface_delay: integer;     primitive;
21    function Propagation_delay:      integer;     primitive;
22    function Interface_hold:         integer;     primitive;
23    function Propagation_hold: integer;     primitive;
24
25
26    module  Network_service;
27
28       ip
29          NSAP:  array [ Side_type ] of NS_primitives( N_provider )
30                 individual queue
31                 fifo delay Interface_delay;
32                        (* delay going out through NSAP *)
33
34    end; (* of module header definition *)
35
36
37    body Network_body for Network_service;
38
39    ip
40    (* internal ip *)
41    Transit:     array [ Side_type ] of NS_primitives( N_user )
42                 individual queue
43                 fifo delay Propagation_delay;
44                        (* propagation delay in Network *)
45
46    var
47      (* Resources *)
48      Propagation_resource:   array [ Side_type ] of resource;
49      Incoming_NSAP_resource:array [ Side_type ] of resource;
50
51    state   Idle, Opening, Open, Close;
52
53    (* Function definition part *)
54    function Opposite (side: Side_type): Side_type;
55      begin
56      if side = B then Opposite := A
57                  else Opposite := B
58      end;
59
60    (* Initialization *)
61    initialize to Idle
62      begin
63      all side: Side_type do
64        begin
65        Propagation_resource[side] := newresource('Propagation resource', 1);
66        Incoming_NSAP_resource[side] := newresource('NSAP resource', 1)
```

```
 67        end
 68      end;
 69
 70
 71    (* DATA transitions: ------------ *)
 72    TRANS
 73    any side: Side_type do
 74    when NSAP[ side ].N_DATA_req (* msg *)
 75    from Open to Same
 76    hold Incoming_NSAP_resource[ side ] for Interface_hold
 77      begin
 78      output Transit[ side ].N_DATA_req(Msg);
 79      end;
 80
 81    TRANS
 82      any side: Side_type do
 83      when Transit[ side ].N_DATA_req  (* msg *)
 84      from Open to Same
 85      hold Propagation_resource [side] for Propagation_hold
 86        begin
 87        output NSAP[ Opposite(side) ].N_DATA_req(Msg);
 88        (* There will be a Hold on the resource Outgoing_NSAP_resource[ side ]
 89           for Interface_hold when this signal arrives at the Network User. *)
 90        end;
 91
 92
 93    (* Control transitions: --------------- *)
 94
 95    TRANS
 96      any side: Side_type do
 97      when NSAP[ side ].N_CONNECT_req
 98      from Idle to Opening
 99        begin
100        output NSAP[ Opposite(side) ].N_CONNECT_ind;
101        end ;
102
103    TRANS
104      any side: Side_type do
105      when NETWORK[ side ].N_ACCEPT_req
106      from Opening to Open
107        begin
108        output NSAP[ Opposite(side) ].N_ACCEPT_ind;
109        end ;
110
111    (* ... OTHER TRANSITIONS FOR NETWORK OPERATIONS ... *)
112
113    end; (* Network_body *)
```