PROTOCOL ENGINEERING FROM ESTELLE SPECIFICATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Graham Wheeler
September 1992

Abstract

The design of efficient, reliable communication protocols has long been an area of active research in computer science and engineering, and will remain so while communication technology continues to evolve, and information becomes increasingly distributed. The formal analysis of protocols is often intractable due to the exponential explosion in the number of global states in a distributed network of asynchronously communicating protocol entities. Simulation is still the predominant technique for performance evaluation of protocols.

This thesis examines the problem of predicting the performance of a multi-layered protocol system from formal specifications in the ISO specification language Estelle. Estelle is a general-purpose Pascal-based language with support for concurrent processes in the form of communicating extended finite-state machines. The use of Estelle has influenced the approach taken; a specification is treated largely as a 'black box' in which only the sequence of state transitions is important, not their internal detail.

The thesis begins with an overview of protocol engineering, particularly performance evaluation and specification. Important parts of the mathematics of discrete-time semi-Markov processes are summarised to assist in understanding the approach to performance evaluation described later.

Not much work has been done in the area of performance prediction from specifications. The idea was first discussed by Rudin, who illustrated it with a simple model based on the global state reachability graph of a set of synchronous communicating FSMs. About the same time Kritzinger proposed a closed multiclass queueing model. Both of these approaches are described, and their respective strengths and weaknesses pointed out.

Two new methods are presented in this thesis. They have been implemented as part of an Estelle-based CASE tool, the *Protocol Engineering Workbench* (*PEW*). In the first approach, a discrete-time semi-Markov chain model is derived from a meta-implementation of a protocol specification. Various ways of using these models predictively are described. The latter is an area ripe for further research.

Like Rudin's, the second approach uses a structure similar to a global-state graph, but many of the limitations of Rudin's approach are overcome, and the models we create using this approach are predictive with great accuracy (complete accuracy under the assumptions used to build the models).

The tool built to explore these methods, the *PEW*, is also described in some detail, and the techniques illustrated with some small examples. The thesis concludes with a discussion of the strengths and weaknesses of the new methods, and possible ways of improving them.

Acknowledgements

This thesis would not have been possible without the support and assistance of the members of the Department of Computer Science of UCT. In particular, I am indebted to my supervisor, Professor Pieter Kritzinger, for keeping me on the right track, and to Professor Riël Smit, my LaTeX mentor. Both their doors were always open to me; I thank them.

Andrew Hutchinson, recently of the Data Network Architectures Laboratory at UCT, listened to many of my ideas and provided me with companionship, constructive criticism and encouragement throughout the course of this thesis.

My parents deserve a special mention for all the love and support they have given me through this time, despite their belief that I should just 'get a normal job'!

This research was supported in part by the Foundation for Research and Development, and in part by Teklogic (Pty) Ltd. The latter company have used the *PEW* for in-house protocol development, and have provided valuable comments and bug reports as well as financial assistance.

This thesis was typeset using Donald Knuth's wonderful TeXtypesetting language and Leslie Lamport's LaTeX macro package. The results would not look nearly as good without these excellent systems, to which so many technical writers are indebted.

Contents

Chapter 1

Introduction

Possibly the greatest impact of the personal computer has been the enormous growth in networking, firstly in the growth of local area networks, and more recently in wide-area internets. This growth has been accompanied by continual technological advances with which the standards-making bodies have struggled to keep up (the Internet excepted). The need for software-engineering techniques and tools for designing and testing communication systems and standards thus continues to grow.

There are numerous sophisticated tools for analysing, monitoring and testing networks, but these usually ignore the details of the protocols (or assume a specific protocol family), and are instead geared towards analysing routing decisions and network topology issues. Part of the difficulty is that protocols are often specified and designed as finite-state machines, and when combined together in networks of asynchronous protocol stacks communicating via unreliable channels, the number of possible states of the system as a whole is usually enormous (the well-known *state explosion* problem). The tools and techniques that have been developed for analysing collections of communicating finite-state machines are primarily aimed at verification, and, to a lesser extent, performance evaluation through simulation.

Some research has been done into using state-space exploration for performance prediction[Rud83, Rud85]. A more common approach to performance evaluation requires building an analytical model of the protocol, such as a closed queueing network[Kri86]. The difficulty with such an approach is that the queuing model must be constructed from the specification of the communicating FSMs, but the model is characterised by a different set of parameters to the protocol - for example, a system might be characterised by a protocol timeout setting and channel bit error rate, but a model of the system is characterised by arrival rates and service times of customers in queues. It is usually not clear how a change in the protocol will affect the model; in particular, if the model is to be used for prediction, it is not clear how its parameter values should be adjusted to predict the performance of a particular configuration of the system.

Thus, performance evaluation directly from protocol specifications is still an open problem, with simulation being the only common approach other than hand-construction of analytical models. This thesis presents two new methods of semi-automatic and automatic performance evaluation and prediction. A software engineering tool, the *Protocol Engineering Workbench* (*PEW*), has been built, incorporating these techniques within a meta-implementation of the ISO specification method *Estelle* (Extended State Transition Language). This has allowed us to experiment with the methods, and compare the results with those obtained from simulation.

The first part of this thesis discusses the field of *protocol engineering.* The main problems involved in engineering communication systems are examined, with particular emphasis on the problems of specification and performance evaluation of protocols.

In the second part, a simple semi-Markov model for a collection of communicating FSMs is introduced, which can be generated automatically from a meta-execution of the protocol specification. The problem of modifying such models to predict performance is discussed, and some basic approaches are described.

An approach similar to a global state space search is then presented. This technique results in a graph structure which is independent of any specific time delay or channel reliability parameter values. Given a specific set of values, the graph can be transformed into a parameterised graph by a labelling algorithm, and the result can be mapped to an imbedded Markov chain model. The latter can be solved to obtain throughput figures for the configuration of the system as described by the parameter values. The process can be repeated for other parameter value sets. The results obtained from this approach are extremely accurate.

The *PEW* is described in the third part, including the restrictions that had to be enforced upon Estelle in order to apply the performance evaluation techniques. The *PEW* is also compared to some other Estelle-based CASE tools.

In the final part two example protocol specifications are examined: the Alternating Bit Protocol, and a subset of LAP-B (the X.25 data link layer). The performance evaluation techniques described above are applied to these protocols using the *PEW*, and the results compared to those obtained by simulation.

We conclude with a discussion of the strengths and weaknesses of the methods, and of the directions in which this work could be taken in the future.

The *PEW* was first described at [KW90], while the modelling techniques were described in [KW91] and [Whe90a], and presented at [KW93].

Part I

Background

Chapter 2

Protocol Engineering

The microcomputer revolution has changed our way of using computers. Large central mainframe sites are increasingly rare, while many people have personal computers on their desks. The disadvantage of this move to distributed computing is the resulting fragmentation of resources and information amongst users. This has resulted in a significant growth in the number and size of computer networks, to the situation today of enormous international networks allowing rapid communication and transfer of information worldwide. Data communications is now one of the primary growth areas in computing and IT.

Related to this is a user-driven move towards *open systems*. ‘Open systems’ is a phrase that is widely used but means different things to different people. One of the best definitions was given by Marshall Rose in [Ros90], who stated that open systems are those in which information is *structured* and *mobile*.

Structure makes information useful and accessible. By encapsulating the structure and providing a standard method of access, it is possible to interwork different applications having different internal structures. To achieve this aim, standards for information access such as Structured Query Language (SQL) have been developed.

Information *mobility* is as important as structure. If information cannot be moved its use is limited; to be shared, it must be transferred. Lying between physical communication media and the applications that use them are the providers of services: the communication algorithms or *protocols*. Protocols have been designed to provide numerous services at different levels, such as error-free in-order delivery of data, network management, directory services, and multimedia conferencing. Higher level services are implemented by protocols which in turn make use of the services provided by lower level protocols.

A good protocol must be flexible, consistent, reliable, efficient, and able to deal with the unexpected. These requirements have led to the study of various aspects of protocols, especially:

- *specification* - describing protocols clearly and unambiguously, so developers in different locations can be sure that their separate implementations are compatible [Boc90, Sun78, vB90];
- *implementation* - turning a specification into an implementation; much study is being done on making this process automatic [Bal85, BGS87, SB90];
- *conformance testing* - ensuring the implementation *does*, in fact, conform to the specification [BRW90, Ray87, Lin90, ABG90];
- *verification* - ensuring that the protocol specification and/or implementation guarantees that good things will happen but bad things will not [Peh90, Sun78, ABG90, Hol90];
- *performance evaluation* - determining the efficiency of the protocol, and 'tuning' it, if possible, to perform better [Con91, Kle88, Lav88, SG78].

As a result, a number of methodologies and tools have been created addressing these problem areas. This thesis describes two techniques for performance evaluation using analytical models and state-space exploration. As analytical models require parameter values, and state-space exploration is computationally intensive, these techniques were implemented in the *PEW*. The *PEW* allows the analysis of Estelle specifications of collections of concurrent communicating extended FSMs.

The remainder of this part of the thesis describes the areas of protocol specification and performance evaluation.

Chapter 3

Protocol Performance Evaluation

There are three major approaches to performance analysis:

- Direct Measurement and Monitoring
- Simulation
- Analytical Modelling

3.0.1 Direct Measurement and Monitoring

The obvious, seemingly ideal method of determining the performance of a system is by measuring the system directly in operation, either by a software monitoring system, or by a non-intrusive monitor in hardware.

Included in the category of software monitors is *accounting software*, commonly available on most multi-user computer systems, and *execution profilers* which determine the performance of specific programs. Software monitoring can be either *event-driven* or *sampled.* In the former approach, the software is modified so that it updates statistics when significant events occur, such as a task switch, or, in the context of computer communications, a frame being sent or received. Sampling, on the other hand, uses a timer-driven interrupt to determine what activity is taking place at regular intervals. Sampling has the advantage that the sampling program is independent of the program under test; in the event-driven approach the program under test must be modified to gather the statistics itself. The frequency of sampling affects the performance of the system; a higher sampling rate causes a higher overhead and thus greater degradation of the overall system performance. On the other hand, a higher sampling rate results in better samples, so there is a trade-off. The event-driven approach adds a fixed overhead to the system, and is thus more accurate.

However, direct monitoring has a number of disadvantages:

- A system must be operational before any performance measures can be made, at which stage it may be too late to rectify any problems that are found;
- Monitoring hardware/software is often very costly;
- Unless the monitoring system is non-intrusive (in hardware), the act of monitoring itself affects the performance;
- It may be difficult to choose and implement a representative workload for the system;
- Monitoring the system may make it unavailable to its normal users.

The advantage of monitoring is that the results come from the real system and thus automatically have some level of credibility.

Even when using simulation or analytical modelling, the monitoring approach has its uses. A simulation or analytical model must be configured with numerical parameters, and the estimation of these parameters is a significant problem. Direct monitoring is one way of obtaining reasonable parameter values.

3.0.2 Simulation

A popular approach to determining performance is through the use of simulation. Simulation offers a high degree of flexibility. It can be used when monitoring is not feasible and the mathematics required in analytical modelling is intractable. The degree of accuracy of a simulation can be increased by making the simulation model more detailed and/or executing the simulation for longer periods of time. Complete control over all the parameters including time is possible, and the system can be observed globally. The primary disadvantages of simulations are the cost and effort involved in constructing accurate simulation models, and the amount of execution time required to obtain reliable results. When the latter is particularly long, parametric studies may not be practical. Nonetheless, the time required to construct a simulation is typically much less than an implementation, and simulations can usually execute much faster than implementations as well.

Simulations usually center around the generation of pseudo-random numbers with particular probability distributions, typically representing arrival times or service times. Because arbitrary probability distributions can be used, simulation is useful in cases where the arrival and service-time distributions do not lend themselves to analysis.

The times generated in a simulation are simulated times, and do not correspond with time in the real world outside of the simulation. Thus simulation is not a 'real-time' process, but occurs within its own time framework. The simulation model is responsible for keeping track of the simulated time, and the scheduling of events in this time frame. This is typically done by maintaining an *event list* which contains a set of events scheduled to occur in the future of the simulation. This list is sorted temporally, and new generated events are added at the appropriate point. The basic operation of a simulation model is thus:

1. remove the event at the head of the event list
2. set the simulation time to be the event time
3. execute the event
4. if there are any new events that will occur in the future as a result of executing the event or changing the time, determine at what time these events should occur, and place them on the event list in the appropriate positions
5. continue at step 1.

Building a simulation model is often very complex, and is mostly done iteratively. The model needs to be sufficiently detailed to capture all the salient characteristics of the system, while remaining simple enough to understand, and omitting redundant aspects. If the model is too detailed, the simulation will execute slower and thus be less useful, while the model will be more time-consuming to construct and more difficult to understand.

Simulation models are typically written using languages specially designed for the purpose, such as SIMSCRIPT or GPSS (General Purpose Simulation System). Often, however, these languages constrain the simulation, both through their limitations and because it may be impossible to obtain measures other than those provided by the language system itself.

An alternative to simulation models is the use of a *meta-implementation* such as that provided by the *PEW* tool described in this thesis. The idea here is to take the actual program/protocol which is to be analysed, and embed it within an environment in which it can execute. In the context of protocols, this means:

- building a compiler and interpreter for the specification language;
- adding surrounding layers to the specification representing the user of the protocols services and the provider of underlying services;
- using the interpreter to execute the compiled specification.

In the *PEW*, a protocol specification should be supplemented by Estelle specifications for an upper layer (called the *user layer*) and a lower layer (called the *provider layer*). The entire system consisting of all of these layers is compiled, and then executed by an interpreter which includes scheduling and time management. This allows a formal specification to be used directly as a form of simulation model, leading to a considerable reduction in effort and no loss in accuracy. The same protocol specification that would be used to generate an implementation is effectively used as a simulation model. The disadvantage is that a specification will execute slower than a simulation which has been carefully constructed to include only the absolute essentials. The advantages, however, are many: not having to construct a simulation model, knowing that the results obtained are consistent with the specification (whereas errors may be introduced in constructing a simulation model), and being able to quickly change a specification to see what happens, make meta-implementations extremely attractive for obtaining performance measures. Furthermore, a meta-implementation can be adapted for other purposes, as this thesis illustrates.

3.0.3 Analytical Modelling

When direct measurement is not possible, a model can be used which captures the salient features of the system under study. This model may be a simulation model (discussed above) or a stochastic model which can be analysed using standard analytical techniques. Generally, analytical models are based on queueing theory, and vary in complexity and thus solution method.

A drawback of analytical models is that they need to be supplied with a set of parameter values; these typically consist of probabilities of routes and service times. Obtaining an appropriate set of parameter values is not always easy; usually an implementation must be measured or a simulation set up to do this. A tool such as the *PEW* is invaluable for this purpose.

The analytical models used for the techniques described in this thesis are based on discrete-time Markov theory. A brief summary of that theory is now given.

Stochastic Processes

Definition 1 *A* **random variable** χ *is a variable whose value is uncertain, but can be characterised by a probability mass function:*

$$p_\chi(x) = P[\chi = x]$$

i.e. $p_\chi(x)$ *is the probability that* χ *has value* x*.*

Definition 2 *A* **discrete** *random variable is a random variable whose value space is countable (but not necessarily finite).*

Definition 3 *A* **stochastic process** *is a family of random variables* $\{\chi(t)\}$ *indexed by the time parameter* t*. If the time index set* {t} *is countable, the process is a* **discrete-time** *process. The possible values or* **states** *that members of* $\{\chi(t)\}$ *can take on constitute its* **state space**. *If the variables are discrete, the process is called a* **chain**.

We restrict our discussion to discrete time chains.

Treating the random variables as a vector $\mathbf{X} = [\chi(t_1), \chi(t_2), \ldots]$, the process can be characterised by its joint probability distribution function:

$$\mathbf{F_X}(\mathbf{x};\mathbf{t}) = P[\chi(t_1) \leq x_1, \ldots, \chi(t_n) \leq x_n]$$

Markov Processes

In 1907 A. A. Markov published a paper in which he described a one-step dependency between the random variables (that is, systems in which the next state depends only on the current state and nothing else). Processes which have this type of dependency are now known as Markov processes. The Markov or 'historyless' property can be described as:

$$P[\chi(t_{n+1}) = x_{n+1} \mid \chi(t_n) = x_n, \chi(t_{n-1}) = x_{n-1}, \ldots, \chi(t_1) = x_1]$$

$$= P[\chi(t_{n+1}) = x_{n+1} \mid \chi(t_n) = x_n]$$

The Markov property requires that the next state can be determined knowing nothing other than the current state, not even how much time has been spent in the current state. This may be less of a restriction than it seems at first. Consider a process with n states in which the next state depends on the two previous states. This system could be replaced by a process with n^2 states, where each state in the new process consists of successive pairs of states from the old process. Any dependence of future behaviour on a finite number of past states, can, in principle, be treated in the same way.

Definition 4 *A* **homogeneous Markov chain** *is one whose probability distributions are stationary with respect to time. That is:*

$$P[\chi(t) \leq x \mid \chi(t_n) = x_n] = P[\chi(t - t_n) \leq x \mid \chi(0) = x_n].$$

The Markov property results in restrictions on the distribution of time spent in a state (the *sojourn time* τ). The next state must be independent of this time, so the distribution of time must satisfy the property:

$$P[\tau \geq s + t \mid \tau \geq t] = P[\tau \geq s]$$

In the case of a discrete-time Markov chain, the only distribution which satisfies this property is the geometric distribution, while for continuous-time the only distribution is the negative exponential distribution.

A homogeneous discrete-time Markov chain can be described by a *stochastic matrix* (or *transition probability matrix*) $\mathbf{P}$ where:

$$p_{ij} = P[X_{n+1} = j \mid X_n = i]$$

That is, p_{ij} gives the probability of j being the next state given that i is the current state.

Definition 5 *A Markov chain is* **irreducible** *if every state can be reached from every other state.*

Definition 6 *Let S_0 denote a subset of the state space S, and $\overline{S_0}$ its complement. Then S_0 is* **closed** *if no single-step transition is possible from any state in S_0 to any state in $\overline{S_0}$. If S_0 consists of a single state x_i, then x_i is called an* **absorbing state**. *A necessary and sufficient condition for x_i to be an absorbing state is that $p_{ii} = 1$.*

Let $f_j^{(m)}$ be the probability of leaving a state x_j and *first* returning to that same state in m steps. Then, the probability of ever returning to state x_j is given by:

$$f_j = \sum_{m=1}^{\infty} f_j^{(m)}$$

Definition 7 *For any state state x_j:*

- *if $f_j = 1$ then x_j is called* **recurrent**; *else*
- *$f_j < 1$ and x_j is called* **transient**.

If the Markov chain can return to state x_j only at steps $\eta, 2\eta, 3\eta, \ldots$ where $\eta \geq 2$ is the largest such integer, then x_j is called *periodic* and has *period* η. If $\eta = 1$ (that is, the process can return to state x_j at any time), then x_j is *aperiodic.*

A Markov chain is *absorbing* if every state in it is either absorbing or transient.

Definition 8 *The* **mean recurrence time** *M_j of state x_j is*

$$M_j = \sum_{m=1}^{\infty} m f_j^{(m)}$$

which is the average number number of steps needed to return to state x_j for the first time after leaving it. If $M_j = \infty$, then x_j is called **recurrent null**, *otherwise it is called* **recurrent nonnull**.

We define:

$$\pi_j^{(m)} = P[\chi_m = x_j]$$

as the probability of finding the Markov chain in state x_j at step m.

We now state an important result[Kle76]:

Theorem 1 *The states of an irreducible discrete-time Markov chain are all of the same type; thus they can be either:*

- *all transient,*
- *all recurrent nonnull, or*

- *all recurrent null.*

If the states are periodic, they all have the same period.

Definition 9 *The* **limiting probability distribution** $\{\pi_j \mid x_j \in S\}$ *of a discrete-time Markov chain is given by:*

$$\pi_j = \lim_{m\to\infty} \pi_j^{(m)}$$

Steady State Distribution

For a proof of the following important theorem see [Kle76].

Theorem 2 *In an irreducible and aperiodic homogeneous Markov chain, the limiting probabilities*

$$\pi_j = \lim_{n\to\infty} \pi_j^{(n)}$$

always exist, and are independent of the initial state probability distribution. Moreover, either

1. *every state x_j is transient or every state x_j is recurrent null in which case $\pi_j = 0$ for all x_j and there exists no* stationary *distribution, or*

2. *every state x_j is recurrent nonnull and then $\pi_j > 0$ for all x_j, in which case the set $\{\pi_j\}$ is a stationary probability distribution, and*

$$\pi_j = \frac{1}{M_j}$$

where M_j is the mean recurrence time of state x_j. In this case the $\{\pi_j\}$ is uniquely determined by the following equations

$$\begin{aligned} \sum_i \pi_i &= 1 & (1) \\ \sum_i \pi_i p_{i,j} &= \pi_j & (2) \end{aligned}$$

If $\mathbf{\Pi}$ is defined as the vector

$$\mathbf{\Pi} = [\pi_0, \pi_1, \pi_2, \ldots]$$

then (2) can be rewritten as:

$$\mathbf{\Pi} = \mathbf{\Pi}\mathbf{P}$$

where **P** is the transition probability matrix. The vector $\mathbf{\Pi}$ is called the *steady-state solution* of the Markov chain.

The states of a recurrent nonnull discrete time Markov chain are said to be *ergodic*, as is the Markov chain itself. If the state space of the Markov chain is finite (which we will always assume it to be), the chain is called *finite*, and if it is irreducible, then it is ergodic.

Clearly, in a steady state, the average time $\tau_i(t)$ spent in state x_i by the Markov chain during a fixed duration t is given by

$$\tau_i(t) = \pi_i t$$

Semi-Markov Processes

In a Markov process a transition occurs at each time interval, although a transition may return the process to the same state. A semi-Markov process is a process in which successive state occupancies are governed by the transition probabilities of a Markov process, but whose sojourn times in each state is described by a random variable that depends on the state currently occupied and the state to which the next transition will be made. At the instants of transition, *the semi-Markov process behaves just like a Markov process.* This is called the *embedded Markov process* of the semi-Markov process.

Consider a discrete time Markov chain with state set $S = \{x_i\}$. Each state x_i has a set of holding (or sojourn) times τ_{ij} which are the times spent in the state before changing to the various states x_j. These holding times are positive integer-valued random variables governed by a probability mass function $h_{ij}(n), n = 1, 2, \ldots$ called the *holding time mass function* for a transition from state x_i to state x_j and where n represents the discrete time variable. That is,

$$h_{ij}(n) = P[\tau_{ij} = n] \quad n = 1, 2, 3, \ldots ; i, j = 1, 2, \ldots, \mid S \mid$$

The *delay time* τ_i of state x_i is the time spent in x_i irrespective of the successor state; this has the probability mass function

$$d_i(n) = P[\tau_i = n] = \sum_{j=1}^{|S|} p_{ij} h_{ij}(n)$$

The mean holding time $\overline{\tau_{ij}}$ is given by:

$$\overline{\tau_{ij}} = \sum_{n=1}^{\infty} n h_{ij}(n)$$

The mean waiting time $\overline{\tau_i}$ is related to the mean holding time $\overline{\tau_{ij}}$ by

$$\overline{\tau_i} = \sum_{j=1}^{|S|} p_{ij} \overline{\tau_{ij}}$$

The Markov chain structure of a semi-Markov process is the same as that of its embedded process. The interval transition probabilities of a semi-Markov process therefore exhibit a unique limiting behaviour within the chain of the imbedded Markov process and we can restrict attention to the latter.

Let $\mathbf{\Phi} = \phi_0, \phi_1, \ldots, \phi_N$ be the vector of probabilities ϕ_j that the semi-Markov process is in state x_j as time $n \to \infty$ and let $\mathbf{\Pi} = \pi_0, \pi_1, \ldots, \pi_N$ be the steady state probability vector of the imbedded Markov process.

It can be shown [How71] that

$$\phi_j = \frac{\pi_j \overline{\tau_j}}{\sum_{i=1}^{|S|} \pi_i \overline{\tau_i}}$$

or

$$\mathbf{\Phi} = \frac{1}{\overline{\tau}} \mathbf{\Pi} \mathbf{M}$$

where

$$\overline{\tau} = \sum_{j=1}^{|S|} \pi_j \overline{\tau_j}$$

and $\mathbf{M}$ is the diagonal matrix $[\overline{\tau_j}]$ of mean delay times.

The average rate at which state x_i will be entered is then given by

$$\frac{\pi_j}{\sum_{i=1}^{|S|} \pi_i \overline{\tau_i}} \tag{3}$$

and the *mean cycle time* c_j, defined as the average time between successive occurences of state x_j is given by the inverse of this.

Chapter 4

The Formal Specification of Protocols

4.1 The Need for Formal Specifications

Protocols are used to communicate across greatly varying distances and between equipment supplied by multiple vendors and having different characteristics. If communication amongst these diverse systems is to be at all possible, they must use compatible protocols, including aspects which may vary between the communicating computers (such as byte-ordering and word size). The protocols are responsible for making each machine appear to the outside world to recognise and obey the same set of commands in a consistent way, much like SQL allows different applications to access different databases in a consistent manner.

In order for protocol implementors to be certain that their implementation will have this high degree of compatibility, they need a clear and complete specification of the protocol, from low-level details such as byte-ordering up to a complete set of responses for every possible event that could occur at any time. However, specifications are often drawn up by standards committees, written in human languages vulnerable to ambiguity, and are rarely 100% complete and correct. Furthermore, such specifications are often a challenge to understand, presented as they are in lengthy documents consisting of many highly cross-referenced paragraphs.

To address this situation, organisations such as IBM[Nas87], the CCITT, and more recently the ISO, have developed *formal* methods for specifying protocols. These *formal description techniques* (FDTs) allow precise, unambiguous specifications. Their use has led naturally to the development of tools and techniques enabling the analysis of formal specifications for correctness, performance, and so on.

Communication protocols lend themselves to being described as extended finite-state machines, or (equivalently) as event-state tables, describing an action and next state for every possible event (message arriving, timer expiring, and so on) in each possible state. Several specification techniques have been developed based on this model.

The discussion below introduces the three FDTs that have been accepted as international standards by the ISO and CCITT. The *PEW* is based on the Estelle FDT, which is described last and in more detail than the others.

4.2 Standard Specification Languages

4.2.1 SDL

SDL[RS82] was designed by the CCITT for specifying functions for telephony-oriented applications. It is part of a family of languages together with the high-level language CHILL and the man-machine language MML. As SDL was designed for specifying systems for telephony, it is particularly suited to large-scale real-time concurrent systems.

An SDL specification can be in a textual or a graphical form. The original form is the graphical form, but the need for a textual form for computer storage and manipulation has led to the development of a programming language-like version corresponding closely to CHILL. A pictorial version also exists; this is a refinement of the graphics version using special telephony-oriented symbols.

SDL has been extensively used by CCITT and by various corporations for defining and documenting operational procedures for concurrent systems.

4.2.2 LOTOS

LOTOS (Language of Temporal Ordering Specification) is based on the idea that systems can be specified by defining the temporal relations between their externally observable interactions. It has been developed particularly for use in OSI (Open System Interconnection) specifications. The underlying formalism of LOTOS is based on *process algebras* (such as Hoare's CSP); this mathematical basis allows LOTOS to be reasoned about mathematically but has also prevented it becoming popular in industry.

4.2.3 Estelle

Estelle (*Extended State Transition Language*) is an FDT developed by the ISO which was standardised in 1989 [ISO89]. The Estelle FDT is less abstract than most others FDTs, and is geared towards implementation. In fact, Estelle is closely related to Pascal in its statements, procedures and functions, with some extensions. Estelle specifications consist of module descriptions, where each module is an extended FSM, and the modules communicate with one another by sending messages asynchronously over FIFO channels. A full discussion of Estelle is beyond the scope of this thesis, but the major features relevant to understanding the remainder of the thesis are outlined.

The Process Structure

An executing Estelle specification consists of a tree of *instances* of generic *modules*. This structure may be dynamic; module instances may instantiate or terminate other module instances at any time. More typically, the root process instantiates a set of processes and interconnects their IPC channels, after which the structure remains static.

Each module represents an extended finite-state machine (EFSM) and consists of a set of states and two sets of state transitions - an *initialisation* set and a *body* set. A random transition from the initialisation set is executed when the module is instantiated, after which the initialisation transitions are ignored, and the execution of the process consists of the selection and execution of transitions from the body set.

The relative behaviour of module instances in the hierarchy is determined by the nesting of their definitions together with their *class qualifiers*. The latter may be any one of `SYSTEMPROCESS`, `SYSTEMACTIVITY`, `PROCESS` or `ACTIVITY`. Processes allow for synchronous parallel execution, while activities allow nondeterministic sequential execution. A module with a class qualifier is called *attributed*; the only non-attributed modules are *inactive* modules, which, as their name suggests, consist only of initialisation parts. Inactive modules may parent only system modules, which may in turn parent only process or activity modules. (System)process module instances may contain either process or activity children, while (system)activity module instances may contain only activity children. A module instance may be created or released only by its parent.

A consequence of the above is that any initial state of a specification defines a fixed number of system instances and communication links between them; this structure, once created, cannot change, since the parent of the systems (if there is more than one system) must be unattributed, and hence inactive. Furthermore, these systems behave fully asynchronously with respect to one another.

Transitions

An Estelle transition consists of zero or more *clauses*, and a transition *body* containing statements which are to be executed when the transition fires. The clauses act as guards, determining whether a transition is enabled or not. They include:

- `FROM` - specifying the state(s) in which the transition begins;
- `TO` - specifying the state in which the transition ends;
- `WHEN` - specifying that a particular message type must be at the head of a particular channel queue;

```
{ Get ACK, discard frame from the buffer, toggle
  the sequence number and return to the ESTAB state }

FROM ACK_Wait TO Normal
  WHEN n.data_response
    PROVIDED (ndata.id=ACK) AND (ndata.seq=send_seq)
      BEGIN
        send_buf_cnt:=send_buf_cnt-1;
        send_seq:=(send_seq+1) MOD 2;
      END;
```

Figure 1: Sample Protocol Transition in Estelle.

- `DELAY` - specifying how long the other clauses must be satisfied for before the transition can actually fire;
- `PROVIDED` - for any other arbitrary Boolean conditions, including constraints on the message specified in the `WHEN` clause, if any; and
- `PRIORITY` - for ordering purposes in the case of multiple fireable transitions.

Transitions may be nested; the clauses of a transition are effectively pushed onto a stack, which is popped only enough to ensure no duplicate clauses upon the next transition. For example, if the second transition began with a `DELAY` clause, and the clause stack contains a `DELAY` clause itself, then all clauses up to and including the `DELAY` clause will be popped off the stack, and the new `DELAY` clause pushed. Any clauses on the stack below the new `DELAY` clause are inherited by the transition. A transition may begin with the `TRANS` keyword, in which case the stack is cleared completely.

Figure 1 shows a sample transition of a protocol written in Estelle.

A transition is said to be *enabled* if its `FROM`, `WHEN` and `PROVIDED` clauses are satisfied; it is said to be *offered for execution* if it has been continuously enabled for the duration specified in its `DELAY` clause, and it is said to be *fired* if it is offered and selected for execution.

At any particular point in time, several children of a systemprocess may be executing transitions, but only one child of a systemactivity may be doing so. Furthermore, the transitions of a parent always have priority over the transitions of the children; this, combined with the fact that variables may only be shared between parent and child processes, guarantees that mutual exclusion is always enforced when shared variables are accessed.

Time in Estelle

The `DELAY` clause, and the concept of time in Estelle, require further elaboration. Firstly it should be noted that a `DELAY` clause may take one or two arguments; a single argument is the same as two identical arguments. The two arguments specify the minimum and maximum delay times. A transition that has all its clauses satisfied can be executed once it has remained in this state for the duration specified by the minimum value. However, it need not necessarily be executed until the duration specified by the maximum value has elapsed, at which time it *must* be executed. Thus the actual delay used is a value in the closed range given by the minimum and maximum delays. Note also that these delays can be arbitrary expressions, and are not necessarily constant. A special delay value, '`*`', is used to indicate that the transition has no maximum delay value but can be indefinitely delayed.

The following is the description of the concept of time in Estelle, as given in the ISO standard ([ISO89] section 5.3.5):

> *Some Estelle transitions may contain a* **delay-clause**... *The intention of the* **delay-clause** *is to indicate that a transition's execution (if it is enabled in a state) should be delayed. Two times are associated with a delay: the minimum time the transition must be delayed and the maximum time it may be delayed. These are initially specified by two values of integer expressions occurring in the* **delay-clause**. *The dynamic change (non-increasing) of these time values with respect to the dynamic change of global situations... relates the progress of time and computation.*
>
> *The computational model for Estelle... is intentionally formulated in time-independent terms: one of the principal assumptions of this model is that nothing is known about the execution time of a transition in a module instance...*
>
> *The only assumptions about time... that are taken into account in this semantic model are that:*
>
> 1. *time progresses as the computation does, and*
> 2. *this progression is uniform with respect to delay values of transitions involved.*
>
> [The second point] *means that given two enabled and delayed transitions in a computation situation, if they remain enabled in the following situation, then their delay values would decrease (if at all) by the same amount... Any constraint which does not contradict the above assumption is admissible...*

We will return to this model of time later in Section 8.5.1, when we discuss the meta-implementation of Estelle specifications in the *PEW*.

Figure 2: The Process Structure of a Three-Layer Point-to-Point Communication System

Inter-process Communication

Another aspect of Estelle which is worth elaborating on is the inter-process communication method. IPC in Estelle is achieved by message passing. Messages are passed asynchronously; that is, the sender is not blocked and made to wait for the receiver. When a message is sent, it is simply appended to the end of a theoretically infinite FIFO queue. This means that although an Estelle specification may consist of a number of FSMs, the system as a whole need not be a finite state system, as an infinite number of channel states can potentially occur. This characteristic obviously has important implications for state-space explorations of Estelle specifications.

Representing a Complete Communication System

Processes in Estelle are arranged in a hierarchy. In order to represent a system consisting of two users communicating over a network using a protocol, a top-level inactive process with five sub-processes could be used: two user processes, connected to two protocol processes, connected in turn to a network process. The top-level process simply initialises the children and establishes their interconnections, after which it remains inactive and the five sub-processes execute concurrently (but asynchronously). For example, Figure 2 shows the process structure for a specification consisting of two user processes communicating with each other over a physical layer process via a data link protocol.

This structure has been used for the examples in this thesis. It is easily generalised to any number of protocol layers, and communicating entities. The implementation of appropriate upper and lower layers is discussed in more detail in Sections 9.1 and 9.2.

4.3 A Note on Terminology

The use of Estelle in the *PEW* has led to some Estelle terms being used interchangeably with more common terms for the same concept. Thus we use 'message' as well as 'interaction', and 'interaction point' as well as 'connection endpoint'.

An exception is that we use the word ‘process’ to refer to an Estelle ‘module instance’, as opposed to the Estelle terminology in which a process is a type of module, and a systemprocess is a refinement of this type. We use the word ‘system’ to refer to the process(es) being studied. That is, in this thesis, we analyse a system (Estelle specification) consisting of a number of processes (executing module instances).

4.4 Conclusion

A number of FDTs have been developed which are certainly improvements on natural languages in terms of precision, but there is no specification language that stands above the rest as an obvious choice. Researchers appreciate the algebraic basis of LOTOS, programmers are more inclined to use Estelle, while telephony engineers have been using SDL for a number of years. The specification languages themselves are not always as useful as they may have seemed to their designers; for example, a number of changes and additions have been suggested for Estelle as users gain experience and discover the shortcomings of the language[Cou87]. There is still much room for further research into this problem.

Estelle was chosen as the specification language on which to base our research for the following reasons:

- it is a non-proprietary international standard;
- it is derived from Pascal, and thus it would be (reasonably) simple to write a compiler (this turned out to be less true than it seemed at first sight!);
- again, due to the Pascal-basis, it is easy for computer scientists and programmers to learn and use;
- being implementation-oriented, it is possible to generate implementations automatically from Estelle specifications.

This is not to say that Estelle turned out to be completely suitable for semiautomatic performance evaluation. There are drawbacks, some of which have been noted in the text. Estelle also does not lend itself well to verification. No doubt better languages will evolve to do the tasks attempted with Estelle in this thesis. The future of protocol specification promises to hold some exciting developments.

Part II

Semi-automatic Performance Evaluation

Chapter 5

Performance Prediction

Both simulation and analysis techniques require the construction of a suitable model of the system being analysed. The aim, however, is to automatically predict the performance of a communication system from its specification, with as little manual intervention as possible. Rudin [Rud83] first suggested this idea, and described a method by which it could be done. Kritzinger [Kri86] proposed a different method, making use of a modified form of the FSM called a *transition relation graph* (*TRG*) to construct a closed multiclass queueing network model.

This chapter describes and critically assesses these approaches, and concludes with a discussion of the problem of parametric modelling; that is, how can the performance of the protocol be predicted as a function of some set of parameters?

5.1 Rudin's Approach

Rudin's model[Rud83] consists of a collection of FSMs representing the states and state transitions of the sender, receiver and channel. The FSMs are synchronous; a message event is generated by each process at each clock interval. A special IDLE message represents a 'do-nothing' event. Asynchronous FSMs were not considered, but Rudin notes that the algorithm would be more complex in this case. Each arc (transition) is labelled with a delay of unity, as well as the probability of being traversed when in the state; the latter would usually be determined by simulation.

The performance technique proposed by Rudin was influenced by existing program verification tools. His idea was to start at some initial state, and then explore every possible sequence of events, creating branches wherever there was more than one possible choice. In this way a tree was created, with each node containing a record of the global state of all the processes and channels at that instant. Exploration of a path was terminated when the cumulative probability of that path becomes smaller than some predetermined cutoff value. The throughput figures were then calculated from a weighted count of the arcs of the tree. In a later paper [Rud85], Rudin assumed that the system is Markov, and used a full global state graph.

Of course, the number of different states grows exponentially with the complexity of the protocol. In Estelle specifications the situation can be worse; due to the semantics of inter-process communication, there are an infinite number of possible channel states in most specifications, unless the specification is explicitly constructed to prevent this.

The method as presented by Rudin has a number of restrictions:

- simulation is still needed to obtain the probabilities;
- communication systems are typically asynchronous, not synchronous;
- all delays are unity; this makes the model very complex if it is to be used to model a system with varying delays associated with transitions (many idle transitions and extra states are required);
- the specification language is model-oriented rather than implementation-oriented;
- the model is parameterised *before* the global state graph is built. Even barring the simulation requirements, this means that Rudin's (later) approach will require much more time and space to determine throughput values than other methods, except for contrived models with very small global state graphs.

These problems may explain why Rudin did not perservere with this approach.

Chapter 6 describes an approach that has some similarity to Rudin's, but has the following advantages:

- a parameter-independent global state graph (called the *behaviour graph*) of the system is built once only;
- the behaviour graph together with a set of parameter values can be processed by a labelling algorithm to build a semi-Markov model of the resulting (parameterised) system;
- the approach can be applied directly to Estelle specifications using the *PEW*. The *PEW* builds the behaviour graph, while a separate graph processing program processes the graph with a set of `OUTPUT` statement reliability values and `DELAY` clause values, and solves the resulting semi-Markov model. Thus, predicting performance for a new set of parameters simply involves modifying a parameter value file and re-executing the graph processor.

State-explosion is the biggest drawback of this approach, but the method is still useful, particularly as deadlock and livelock problems are detected as a byproduct. Such problems are easy to detect by examining the throughput rates of transitions calculated by the graph processor for some set of parameter values. The method thus also can identify problems in a protocol specification that occur only under certain sets of parameter values but not under others.

5.2 Kritzinger's Approach

About the same time as Rudin's work, Kritzinger [Kri86, Kri89] proposed a performance analysis methodology of converting a formal specification into a closed multiclass queueing network. The routes taken by customers through this network can be determined semi-automatically from the static specification, but simulation or similar is still required to determine the parameters describing the service time and route probabilities for each customer class. Once the queuing network has been built, it can be solved to determine the throughput rates of the various customer classes. Kritzinger specified the queueing model using the language SNAP/L and solved it with the MicroSNAP tool which implements the Mean-Value Analysis (MVA) algorithm [RL80].

The model used by Kritzinger is based on one used for verification[LS84]. Consider I protocol entities $P_1, P_2, \ldots, P_I$ and K channels $C_1, \ldots, C_K$. Let S_i be the set of states of P_i and M_{ik} be the set of messages that P_i can send into C_k.

The dynamics of the components are described by entity events and channel events. The latter model various types of channel errors and change only the state of the channels, not protocols, and are not considered any further.

There are three types of entity events:

Send events : Let $t_i(r, s, -m)$ denote the event of P_i sending a message m into channel C_k where $m \in M_ik$ and C_k is in the outgoing channel set of P_i. The send event is enabled when P_i is in state r. After the event P_i is in state s and m has been appended to the end of the message queue in C_k.

Receive events : Let $t_i(r, s, +m)$ denote the event of P_i receiving a message m from channel C_k where $m \in M_ik$ and C_k is in the incoming channel set of P_i. The receive event is enabled when P_i is in state r and m is at the head of the message queue of channel C_k. After the event P_i is in state s and m is deleted from the channel queue.

Internal events : Let $t_i(r, s, \alpha)$ denote an internal event of P_i where α is a special symbol indicating the abscence of a message. The event is enabled in state r and results in P_i ending in state s. Internal events are used to model timeout events internal to P_i as well as interactions between P_i and its local user.

The set of events for P_i is denoted T_i. Each event $t_i(r, s, x)$ (where x is any of $+m$, $-m$, or α), is associated with a probability $p_i(r, s, x)$. A measure of the performance of the protocol would typically be the rate at which some send event is executed.

Kritzinger assumes that all protocol entities contend for service at a single processor, and considers this model as a closed multiclass queueing system with two service centres. The one service center is a Processor Sharing type which models the processor, and the other is a pure delay server which models the delays experienced in the channels as a result of receive events. Customers in the network are instances of the protocol entities P_i, and each take on a distinct class associated with each distinct entity event causing a state transition. Each distinct entity event (or class) is classified either as *active* (requiring processor time) or a *delay*; these classifications determine which server they will request service from. Typically, send events are active, receive events are delays, and internal events may be either. Events which are both active and delays must be decomposed into two separate events with a new intermediate state. In any case, each entity event $t_i(r, s, x)$ is associated with an expected service time duration $\mu^{-1}(r, s, x)$.

The only assumptions required to solve this model as a closed multiclass queueing network are that:

- the probabilities $p_i(r, s, x)$ are independent for all r, s and x, and
- the time $\mu^{-1}(r, s, x)$ associated with an event is independent of the time associated with any other event.

Kritzinger generalised the method to multiple layers executing concurrently on a single or multiprocessor system. Each processor has a corresponding processor server of the processor-sharing type. Each layer of the protocol, or separate process, makes up a separate closed chain, of which there may be several.

The method can easily be extended. For example, not only the throughput of the protocol, but also the processor utilisation, may be determined. By adding a server representing disk storage, large file transfers could be modeled and processor, disk and network utilisation determined.

In deriving the queueing network, Kritzinger makes use of a *transition relation graph* (TRG), which is a graph derived from an FSM by creating nodes corresponding to the state transitions of the FSM, and connecting these with arcs which describe the possible sequences of transitions that can occur and their relative probabilities and delays. This structure is closely related to the stochastic model described in Chapter 6.

Kritzinger's method is a useful tool for automated analysis, but suffers from drawbacks pointed out by Conway [Con88, Con91]. Firstly, the problem of determining model parameter values: only the structure of the model can be determined statically from a specification. Simulation or some similar technique must still be used, which could be used to obtain the throughput measurements in the first instance (this criticism applies to Rudin's method as well).

Conway also argues that the model fails to take into account causal *linkages* between the layers, as each process executes asynchronously with respect to the others. This second criticism is misleading: if parameters are obtained from a simulation (for example), in which all of the necessary layers were represented, the parameters would inherently reflect the linkage relationships. On the other hand, these linkages make the relationships between the modelled system and model parameter values more complex and difficult to use for prediction.

5.3 Parametric Prediction

When investigating a protocol's performance, we usually would like to ask "What if?" questions. How would the performance be altered if we shortened the timeout? What would happen if the line errors increased? To answer these questions, we have to undertake parametric studies. The parameters we wish to vary are typically the time required by, or probability of, particular events, or the size of buffers.

Ideally we would like to be able to predict the answers to such questions without having to redo all the work for each case, but rather by use of some model or algorithm with which we can simply specify the range of cases we are interested in and have the results produced automatically in a reasonable period of time. It seems very unlikely that this will ever be possible to do in general. The contradictions of the well-known Halting Problem should be equally applicable to such a 'universal' performance prediction tool. The methods described above certainly fall very far short of this.

Rudin's method essentially obtains a performance prediction as a by-product of verification (that is, global state space searching). Once constructed, the model is restricted in its predictive capabilities. The example Rudin uses is trivial, allowing all arc probabilities to be described as functions of the channel reliability, without a single simulation being required. A real protocol will never be this simple, and thus simulations must be performed for each set of parameters and the arc probabilities measured. This achieves little, as the performance could be determined from all the simulations without ever constructing a model. Kritzinger's approach, while computationally much simpler, suffers from the same problem.

In this thesis we describe Two new methods for performance prediction are presented in this thesis, one related to Kritzinger's TRG representation, and the other, like Rudin's, to state-space searching. Both of these approaches can be computationally intensive, but they do result in predictive models that are more useful than the earlier attempts.

Chapter 6

A Simple Performance Model

In this chapter we show how a discrete-time semi-Markov chain model can be derived from an execution trace of a set of communicating FSMs. The method is used in the *PEW* to automatically generate such models from the meta-implementation of an Estelle specification. A separate tool uses Gaussian reduction to solve the imbedded Markov chain, and then adjusts the results to take into account the holding times, finally producing a set of throughput figures for the various transitions of processes in the specification. These throughput figures are consistent with the throughput figures obtained directly through measurement during the meta-implementation.

Having shown how a model can be derived (and parameterised) automatically from a specification, we turn to the problem of using the model for performance prediction. We restrict attention to two types of independent variables, namely channel reliabilities and transition delays. There are numerous possible approaches here, and this area is one in which much additional research could be done. We describe the approaches we have used to date; the conclusion of the thesis suggests some other approaches to exploring this problem.

Using the *PEW*, we have constructed predictive models of several Estelle specifications, some of which are described in Part IV.

6.1 Initial Attempts

The *PEW* was built as a tool for developing, testing, and debugging Estelle specifications, and the first performance evaluation capabilities that were introduced involved the automatic tabulation of user-defined expressions for a set of executions of the system, each time incrementing some independent `CONST` definition in the specification by a fixed value. For example, the user could define an expression for throughput, and, using a `CONST` to specify the delay of a time-out transition, have the *PEW* tabulate a set of (timeout, throughput) pairs over some linear range of timeout values.

Our next aim was to have the *PEW* automatically build a stochastic model of the system. The initial approach followed Kritzinger[1]: we derived a TRG from an execution of the system, and then generated MicroSNAP code for corresponding queuing network model. There were some problems with this approach, particularly in determining to which servers the transitions (entity events) should be allocated.

It soon became apparent that a simpler model could be created based on the TRG. To understand this model, we first need to briefly describe how specifications are executed using the meta-implementation provided by the *PEW*.

6.2 Execution Traces

The execution of a system is controlled by a scheduler, which determines which transitions are ready to fire, and then passes control to them. If transitions are atomic (as they are in Estelle), and assuming that the scheduler is operating on a sequential computer (as is the case with the *PEW*), then in reality the scheduler must execute transitions in sequence, even though the FSMs are in principle operating concurrently. This is not a problem - the scheduler evaluates the transition sets and selects a group for 'concurrent' execution. Each transition in the group is then executed in turn, after which the global system clock is updated and the whole process repeated. Even though transition execution is no longer concurrent, the transition clauses are all evaluated in the same state, and the semantics are preserved.

A practical implication of this is that a particular execution can be described by a sequential *execution trace*, listing the time at which each transition fired and the order in which they fired. The latter is important - transitions which in principle fired at the same time now have a sequential order even though they are associated with the same execution time.

The execution trace can contain considerably more information. For example, the *PEW* can also note the effect that executing each transition had on the clauses of all other transitions. This information can be useful with the stochastic models described in this chapter and is essential for the behaviour graph model described in the next chapter.

6.3 The Model

Consider a collection C of I communicating finite-state machines $M_1, \ldots, M_I$. Each FSM M_i has a set of transitions $S_i = \{x_{i1}, \ldots, x_{in_i}\}$.

[1]Many other ways were considered, including generating constraint-based systems, state exploration (which led to the behaviour graph technique), perturbation analysis and others. In most cases we encountered difficulties, such as lacking sufficient information to solve the sets of constraints, or being unable to determine any way of automating the approach. It is possible that some of these approaches would work much better with languages more constrained than Estelle. As it is we are forced to impose some restrictions on Estelle specifications before the behaviour graph method can be used.

Assuming such model of execution described in the previous section, we can characterise a particular execution of M_i of duration M as a sequence E_i of N_i^M transition executions:

$$E_i = t_{\alpha_i(1)}, t_{\alpha_i(2)}, \ldots, t_{\alpha_i(N_i^M)}$$

where:[2]

$$\alpha_i(1) < \alpha_i(2) < \ldots < \alpha_i(N_i^M)$$

are the instants in time at which the transitions occur, and

$$t_{\alpha_i(1)}, \ldots, t_{\alpha_i(N_i^M)} \in S_i$$

We define the *delay sequence* D_i to be the time intervals between transition executions:

$$D_{ij} = \alpha_i(j+1) - \alpha_i(j)$$

The execution trace thus described can be easily mapped to a discrete-time semi-Markov chain. Each transition maps to a state of the chain, and the stochastic matrix $\mathbf{P}$ of the imbedded Markov chain is easily obtained from a simple count of the number of occurrences of each unique transition sequence pair. The delay or waiting times $\{\tau_k\}$ are obtained from the means of all D_{ij}'s where $t_{\alpha_i(j)} = k$. The holding times τ_{ij} can be obtained similarly.

The core of the technique, as implemented in the *PEW*, is as follows. The specification is executed for a reasonable amout of time. An execution trace is created, and from this the *transition sequence count matrix* and *transition sequence delay matrix* are produced. The former is a count of the occurrence of each unique consecutive transition pair, from which the stochastic matrix describing the embedded Markov process can be derived by dividing each entry by the sum of the values in its column. The second matrix corresponds to the set of measured holding times for each unique pair of transitions. These two matrices contain sufficient information to determine the throughput of the system analytically.

In particular, if the stochastic matrix P has steady state solution $\{\pi_i\}$, and $\{\tau_i\}$ is the set of waiting times (easily determined from $\{\pi_i\}$ and the holding times $\{\tau_{ij}\}$), then the throughput of transition i is given by:

$$Throughput_i = \frac{\pi_i}{\sum_i \pi_i \tau_i}$$

[2]Of course, it is possible, as we mentioned earlier, that transitions can occur at the same time, in which case the strong ordering of the time instants does not hold. We can overcome this in two ways: we can either assume an infinitesimally small delay between 'consecutive' but simultaneous transitions, or we can aggregate groups of simultaneous transitions into new 'entity events' and work with these. However, the method still works even if elements of the delay sequence defined below are zero, so we do not actually have to do anything.

6.4 A Simple Example

Consider the following execution sequence and delay sequence:

$$T = t_1, t_2, t_1, t_2, t_3, t_2, t_1, t_3, t_3, t_2, \ldots$$

$$D = 4, 6, 3, 4, 8, 2, 2, 6, 6, 4 \ldots$$

In each case we assume that the sequences repeat themselves over and over. The execution count matrix is:

$$\begin{pmatrix} 0 & 2 & 1 \\ 3 & 0 & 1 \\ 0 & 2 & 1 \end{pmatrix}$$

and thus the stochastic matrix is:

$$\begin{pmatrix} 0 & \frac{2}{3} & \frac{1}{3} \\ \frac{3}{4} & 0 & \frac{1}{4} \\ 0 & \frac{2}{3} & \frac{1}{3} \end{pmatrix}$$

The holding time matrix is:

$$\begin{pmatrix} - & 5 & 6 \\ 3 & - & 8 \\ - & 3 & 6 \end{pmatrix}$$

The mean delay times are thus:

$$\begin{aligned} \tau_1 &= \tfrac{2}{3}5 + \tfrac{1}{3}6 &= \frac{16}{3} \\ \tau_2 &= \tfrac{3}{4}3 + \tfrac{1}{4}8 &= \frac{17}{4} \\ \tau_3 &= \tfrac{2}{3}3 + \tfrac{1}{3}6 &= 4 \end{aligned}$$

If we solve the matrix, we obtain $\pi_1 = \pi_3 = \frac{3}{10}$, while $\pi_2 = \frac{2}{5}$.
From the mean delays:

$$\sum_i \pi_i \tau_i = \frac{3}{10}\frac{16}{3} + \frac{2}{5}\frac{17}{4} + \frac{3}{10}4 = 4.5$$

The throughputs of the transitions with respect to time are thus:

$$Throughput_1 = Throughput_3 = \frac{\frac{3}{10}}{4.5} = \frac{1}{15}$$

$$Throughput_2 = \frac{\frac{17}{4}}{4.5} = \frac{4}{45}$$

6.5 Performance Prediction

Although simple, this model gives accurate throughput results and is independent of details of delay distributions. The results obviously agree with those obtained from a simple count of the number of occurrences of each transition in the trace divided by the execution time, and are thus consistent with the execution from which the trace was taken. Of course, the throughput figures can be obtained directly from the execution trace without bothering with building a model - the same problem that affects Kritzinger's and Rudin's approaches. In order to be really useful, we should be able to use the model to predict the performance of different configurations of the system.

It may be asked what use a model is that is built upon Markov theory and yet violates several assumptions about such models, including the Markov property and the constraints on delay distributions? Certainly it is true that using Markov theory to make *predictions* with this model will always be approximate, as these constraints are not satisfied. Nontheless, this offers a first approach to prediction; we describe it and others shortly.

The stochastic matrix summarises a considerable amount of information about the possible sequences of events in the system (the execution trace even more so). With some experimentation, we can often determine relationships between these possible sequences and parameters of the system. We can then predict what the stochastic matrix will be for a modified system with different parameter values.

Making this more precise, consider a (sufficiently long) execution of a system using a set of parameter values V which results in an execution trace T_V. This execution trace maps to a stochastic matrix P_V with elements p_{ij}^V. Each transition t_i has mean delay τ_i^V. We would like to determine two sets of functions f_{ij} and g_i:

$$p_{ij}^V = f_{ij}(V)$$

$$\tau_i^V = g_i(V)$$

That is, given any set of parameter values V, can we determine the stochastic matrix and holding times corresponding to the steady state of the system under these parameters?

We restrict our set of modifiable parameters V to be the delays associated with transitions, and the reliability of message transmissions in transitions (this is more specific than using channel reliability). Message transmissions that were 100% reliable in the original system are not included in V.

Consider what happens when the reliability of a message transmission is modified. Typically, the sequence of transitions will be different, as some error recovery will take place. As the reliability decreases, the execution rate of the error recovery transitions increases and the execution rate of the successful transmission part of the FSM decreases. This will be reflected in the stochastic matrix.

There may also be some changes in the holding times associated with each transition, as these times reflect measured times between transition executions, not how long the transitions themselves took to execute. Unfortunately, the cause (unreliable transmission in transition t_i) and effect (error recovery) are usually going to be more than one step apart, and the effect on the probabilities is often going to be more complex than just changing the row in the stochastic matrix corresponding to t_i. If a delay parameter in the system is changed, the results are often more complex again.

There are a number of heuristic techniques we have used in addressing this problems. We describe some of them below.

6.5.1 Modifying Delays

The holding time matrix $\{\tau_{ij}\}$ contains considerably more information about the delays between transitions than just the mean delay times $\{\tau_i\}$. This means the second set of mapping functions can have the form:

$$\tau_{ij}^V = g_{ij}(V)$$

If we can determine g_{ij}, we can predict a new holding time matrix, and determine the new mean transition delays.

A `DELAY` clause in Estelle can specify a lower and upper bound, which may be identical. Note that the lower bound indicates a delay that elapses *before* the transition fires rather than after. Thus, in the simplest case of modifying delays we modify a column in the holding time matrix rather than a row.

Predicting a new holding time matrix when the `DELAY` clause value of transition i is changed can be done in several ways; for example:

- column i in the holding time matrix (i.e. the holding times of transitions that preceded the `DELAY` transition) can simply be adjusted by the difference between the new value and the value used in building the execution trace;
- by examining several $(\mathbf{P}, \{\tau_{ij}\})$ pairs corresponding to execution traces for different parameter sets V, the user can estimate a suitable mapping function;
- interpolation between two or more $\{\tau_{ij}\}$ for different parameter sets V can be used.

Interpolation is described more in the next section; the second approach can often be aided by focusing attention on a subset of the possible execution traces; this is discussed shortly.

6.5.2 Using Interpolation

A simple approach which can be fully automated is to generate a set of models and then use interpolation between the different matrices to predict the intermediate results. For example, to predict models corresponding to a range of parameter values in which one particular parameter is varied through a linear sequence of N values, we could perform three executions (for the two extreme values and the average value), and then interpolate to obtain mapping functions f_{ij} and g_{ij} for the stochastic and holding time matrices.

6.5.3 Focusing Attention

The system being studied can often be simplified considerably, and this simplification in turn simplifies the problem of determining mapping functions. There are several ways to do this.

We can restrict our attention to a subset of the transitions in a system with no loss of generality (and, because of the interleaved sequential execution model, these transitions may be from any mix of component FSMs of the system). In this way the relationships between system parameters and the stochastic and delay matrices can often be made much more obvious.

There are some guidelines that should be followed when selecting a transition subset:

- at least one transition must reflect the throughput measure we are interested in;
- all transitions with delay parameters in V should be included;
- all transitions with unreliable message transmissions should be included;
- transitions which can enable `DELAY`ed transitions should be included.

Transitions which do not have any future effects with regard to losses or delays can always be ignored. Often it is possible to concentrate on a small subset of the transitions in the process or system. For example, in the Alternating Bit protocol, we considered only three transitions: sending an interaction to the provider, receiving an acknowledgement, and timing out and resending. The resulting model produces good predictions of changes in throughput from a single measured TRG and delay matrix, as will be shown in Chapter 11.

The transitions which enable `DELAY`ed transitions can often indicate relationships between the `DELAY` clause values and the holding times. As an alternative to including an enabling transition in the trace, the enablement of the `DELAY`ed transition could be associated with the `DELAY` transition which was executed immediately preceding the enabling transition (of course, this cannot be done if the enabling transition itself is a `DELAY`ed transition!) For example, if we determine that 30% of the time that `DELAY`ed transition d_1 was enabled, its clock started executing immediately after the execution of `DELAY`ed transition d_2. Then, if the delay associated with d_2 is lengthened, we can predict that 30% of the executions of d_1 will be preceded by a longer delay, and we may be able to determine exactly which holding times are affected by the change.

A similar approach is possible with unreliable `OUTPUT`s. A transition that has an unreliable `OUTPUT` can easily be converted to three transitions, each of the two new high-priority transitions simply clearing an enabling flag (each has its own flag). The unreliable `OUTPUT` transition sets one of the two flags depending on whether the transmission succeeds or fails. The `OUTPUT` transition can then be excluded from the trace, and the two new transitions included. This can assist in determining how to modify the matrices for a change in the reliability of the `OUTPUT` statement.

Another simplification is to construct the system so that there are two nodes, one sending data and the other receiving, and then concentrate only on the sender. Yet another approach is to use a loopback test model, in which the protocol communicates with itself (that is, the physical layer simply echoes all messages back to the sender). Furthermore, if we are interested in protocol throughput we can often the connection establishment and release part of connection-oriented protocols.

6.5.4 Other Techniques

Other techniques we have used include:

- using deterministic scheduling of transitions rather than the usual non-deterministic scheduling used in Estelle;
- using channel blocking to keep causes and effects close together;
- gathering and analysing information about the effects the execution of each transition had on the clauses of other transitions.

Several of the techniques described in this chapter are used in the examples in Part IV.

6.6 Conclusion

Figure 3 summarises the modelling technique. A specification of a system is executed by the *PEW*. The *PEW* builds a transition sequence count matrix and transition delay matrix corresponding to this execution. This may be for each process as a whole, the system as a whole, or for subsets of transitions within the system.

The matrices thus obtained can be used as the basis of a model as outlined above, and new matrices predicted for different value sets V. A separate program reads files containing stochastic matrices and delay matrices, solves the Markov chain, computes the mean delays of the transitions, and then calculates the throughput rates.

After using the *PEW* to assist in deriving a suitable set of mapping functions f_{ij} and g_{ij}, a short program is typically written to generate the data files containing the matrix instances and execute the model solver program for a range of different parameter value sets.

The key to determining the mapping functions is to simplify the behaviour of the system being studied, to the point at which the resulting model can be comprehended and used for prediction, without simplifying to the point at which the perceived relationships do not actually hold or the

Figure 3: Summary of the Modelling Technique

model no longer bears any relation to the original protocol. At present this is a rather *ad hoc* process. We usually generate between three to five different execution traces of the system for selective sets of parameter values, and use the information from these to build a model. In some case our mapping functions were derived using interpolation from the execution traces, while in other cases they were created by reasoned argument and examination of the behaviour of the protocol. The method is not yet a fully automatic performance prediction technique, but provides a useful set of techniques with which to approach the problem. If interpolation were used in deriving the mapping functions in every case, the process could be completely automated.

The heuristic techniques we have outlined in this chapter for simplifying the problem of determining a suitable set of mapping functions represent early steps in the direction of automatic analytical performance prediction; there is scope for much further research into this problem. The conclusion of the thesis describes ways in which the problem could be explored further.

Chapter 7

Behaviour Graphs

In Section 5.1 Rudin's method of predicting performance from a specification was described, and some limitations pointed out, including:

- simulation is still needed to obtain the branch probabilities;
- the model must be synchronous with delays of unity;
- the global state graph method suffers from a state explosion problem;
- the model has limited, if any, predictive power.

A performance prediction method was developed independently as part of the *PEW*. This method is similar to Rudin's, but has considerably more predictive power. Furthermore, the use of Estelle as the specification language means that specifications are implementation-oriented, which is a major advantage. The technique described here can be used to predict the performance of any system described by an Estelle specification, under certain restrictions which are noted.

The method, as implemented in the *PEW*, involves the construction of a directed graph which describes all possible execution traces that may be described by the system, independent of any specific `DELAY` clause values or unreliable `OUTPUT` probabilities. This graph is called the *behaviour graph* of the system. Given a set of values for the `DELAY` clauses and unreliable `OUTPUT`s, a labelling algorithm is described which is used to derive a *parameterised behaviour graph* from the behaviour graph. The latter graph describes a semi-Markov chain model which accurately describes the stochastic behavour of the original system. By repeatedly applying the labelling algorithm with different parameter value sets and solving the derived imbedded Markov models, any number of performance predictions for the system can be obtained.

The biggest drawback to this method is the state explosion problem. Estelle does not help: an arbitrary Estelle specification is more likely to describe a system with an infinite number of states than a finite one (the unboundedness of message queues already implies this). The technique hinges on the number of global states of the system being finite (and preferably not too large for the method to be tractable!) To achieve this, several restrictions on the set of admissible Estelle specifications must be imposed, some to ensure the number of states is finite and some to reduce the number of states further. We now examine the factors which influence the size of the graph, and consider how to lessen their effect. At the same time we assume that the values of `DELAY` clause times and `OUTPUT` unreliabilities are unknown.

7.1 Nondeterminism in Estelle Systems

For the following discussion it is sufficient to consider a behaviour graph to be similar to a global state graph. The difference between these two structures will be clarified later. It is worth noting now, however, that the size of the behaviour graph is more dependent on the number of nondeterministic choices in the system than it is on the number of global states.

Nondeterminism is an important issue in building a global state graph. Nondeterministic choices of the sequences of events result in branches in the graph to different successor states depending on the choice. There are several situations in which nondeterminism can occur in executing a system specified in Estelle.

7.1.1 Scheduling Decisions

The Estelle standard specifies that the selection of which transitions to fire of those that are enabled is nondeterministic. If there were n possible choices, then the node in the global state graph corresponding to the current state would have n arcs leaving it, each going to a state corresponding to one of the n choices.

In practice, protocol implementors would not use random choice of the available possibilities to determine what to do next; their programs would be written in a deterministic way (although there may be cases, such as when using interrupt-driven event handlers, that a sequential process may exhibit non-determinism).

The number of branches in the global state graph can thus be reduced by introducing the restriction:

Restriction 1 *Transition scheduling is deterministic.*

This restriction is not required for the method to be applicable; it simply helps to reduce the size of the graph, thus reducing the size of the problem.

At present, the *PEW* automatically switches to deterministic scheduling when building a behaviour graph (although for normal executions scheduling is nondeterministic unless the user specifies otherwise). The *PEW*'s scheduler will select the first enabled transition in order of appearance in the process, after first taking into account the priorities of all the enabled transitions.

7.1.2 Loss Decisions

If unreliable `OUTPUT`s are being used (as allowed by the *PEW*), then the execution of such a statement is a cause of nondeterminism; it results in a binary branch in the global state graph. The probability of loss tells which of the two possible futures (next states) is more likely, but other than that, *the futures are independent of the specific probability value.* Thus it does not matter that the probability is unknown, as we have assumed above, provided we know that the `OUTPUT` is unreliable. By convention, the *PEW* treats left branches in the graph as 'no loss' branches, and right branches as losses. The only restriction imposed on unreliable `OUTPUT`s is:

Restriction 2 *No transition has more than one unreliable* `OUTPUT` *statement.*

Without this restriction, multiple branches may occur in the graph all corresponding to part of the execution of a single transition. While this situation could be handled, it makes the implementation (and description) of the behaviour graph method much more complex.

Another restriction necessary only to keep the implementation simple is:

Restriction 3 *The specification does not generate random events other than message losses.*

This restriction states that the specification cannot have internal random number generation routines controlling its behaviour. The *PEW* is unable to detect and handle such cases, and will fail to build a graph if there are causes of nondeterminism in the specification beyond its control.

7.1.3 Delay Decisions

The execution model used by the *PEW* is one of *maximal progress.* This means that if the *PEW* can execute a transition, it will. Only if there are no transitions available for execution will the *PEW* advance the system clock and examine `DELAY`ed transitions.

Given the restrictions thus far, and ignoring losses for the time being, it should be apparent that the sequence of executed transitions in the system will be deterministic, until the scheduler must choose between `DELAY` transitions. If the delay values are unknown (as we have assumed), then the scheduler cannot decide which delayed transition to fire, as it cannot determine which of the delayed transitions would occur earliest. If there are n `DELAY` transitions that have all other clauses satisfied, then there are n possible choices. All of these possible futures can be included in the graph, by creating n arcs out of the node at which the scheduler makes its choice, each one corresponding to a particular choice and thus a particular future.

Figure 4: A Producer-Consumer Specification in Estelle

7.1.4 Summary

The set of restrictions described above ensures that the execution of a system proceeds deterministically except when unreliable `OUTPUT` statements are executed or when the scheduler must choose a `DELAY`ed transition, and that at each of these points it is possible to explore a finite set of possible futures. The number of possible futures can be reduced further by minimising the number of unreliable `OUTPUT` statements and the number of transitions with `DELAY` clauses in the specification.

7.2 Creating a Behaviour Graph

We assume that the restrictions listed above are satisfies, and now describe the behaviour graph structure and how it is constructed. The method is illustrated with a simple Producer-Consumer example, shown in Figure 4. The producer has two transitions which send messages to the consumer, at certain delay intervals. The consumer consumes these messages as they arrive. The first `OUTPUT` transition in the producer is unreliable, while the second is completely reliable. This is the simplest example that illustrates the necessary concepts.

7.2.1 Graph Structure

At the point when a specification begins execution it is in a state known as the *pre-initial state.* If the restrictions above are satisfied, an *execution tree* can be derived from a specification. The execution tree consists of all possible execution traces. Branches occur in the tree whenever a transition with an unreliable `OUTPUT` is executed, or the scheduler had to choose a `DELAY`ed transition. Each node of the tree contains a (possibly empty) sequence of transitions listed in order of execution (recall the ordered execution described in Section 8.5.1). Each node can also be labelled with the time at which the transition sequence contained within the node was executed, although this is not needed.

Definition 10 *An* execution tree *is an ordered tree rooted in the preinitial state. Each node N of the tree contains a (possibly empty) sequence of transitions $t_{N,1}, \ldots, t_{N,n_N}$, which is a deterministic sequence of transition firings with negligible delays in between. The end of each sequence indicates a point at which either:*

- *transition t_{N,n_N} contained an unreliable* `OUTPUT` *statement, in which case the node will have two arcs emerging from it, the left indicating the transmission was reliable and the right indicating that is was not; or*

Figure 5: Part of the Execution Tree of the Producer-Consumer System

- *the scheduler had to choose a* `DELAY`*ed transition, in which case the first transition in each child node's sequence is a* `DELAY` *transition which has all other clauses satisfied at that point. The number of arcs emerging from such a node depends on how many* `DELAY` *transitions the scheduler had to choose from, but will be at least one.*

When the producer-consumer system is executed, after a short while the execution tree will resemble that shown in Figure 5. The nodes of the tree contain the transition sequences that occurred. The solid arcs are branches due to unreliable `OUTPUT` statements, while the dotted arcs represent those due to `DELAY` transitions. It is easy to see that this tree is repetitive[1].

We define **children**(N) to be the number of children of node N, and **child**(N, i) to be the i'th child of N.

The construction of an execution tree can be terminated when all paths through the tree have leaf nodes representing global states which occur at least once somewhere else in the tree. However, as the tree does not contain information about global states, a more appropriate definition of *node-equivalence* is the following:

Definition 11 *Two nodes N and M of an execution tree are* node-equivalent *iff:*

- $n_M = n_N$
- $t_{N,i} = t_{M,i}$ *for each* $i = 1, \ldots, n_M$
- **children**(N) =**children**(M)

[1]There are four different types of nodes: empty, those with $p1$ only, those with $c1$ only, and those with the sequence $p2, c2$. The subtree rooted in any particular node is the same as all others rooted in nodes of the same type.

Figure 6: Behaviour Graph of the Producer-Consumer System

- **child**(N, i) *is node-equivalent to* **child**(M, i) *for each* $i = 1, \ldots,$**children**(N).

When a leaf node is found that is equivalent to some other node, all arcs entering the leaf node can be altered to point to the equivalent node, and the leaf node can be pruned. The end result of applying this process repeatedly until no further leaf nodes exist and each remaining node is unique (that is, not node-equivalent to any other), is a cyclic graph which we call the *behaviour graph* of the system.

Definition 12 *A* behaviour graph *is a graph derived from an execution tree in which each arc ending at a node* N *which is* node-equivalent *to some other node* M *of smaller depth in the tree, is replaced by an arc with the same initial point but in which the end point is* M *rather than* N*. After this replacement, there will be no arcs entering* N *and it can be removed from the graph. This process is repeated until there are no further pairs of node-equivalent nodes in the graph.*

The behaviour graph derived from the execution tree of the Producer-Consumer system is shown in Figure 6.

Regardless of what the loss probabilities and delays actually are, this graph contains all the salient information about the possible execution traces that can occur in the system (given the restrictions such as deterministic scheduling). It should be immediately apparent that such a graph has applications in performance analysis and verification. An empty node which loops back to itself, for example, indicates possible deadlock[2].

[2]This situation cannot arise while building the behaviour graph, but can arise in the labelling process described later.

7.2.2 The Graph Building Algorithm

To use the recursive definition of node-equivalence in practice, a cutoff depth should be specified after which it is assumed that all further descendants are also equivalent. This obviously carries a risk but it simplifies the implementation and reduces the memory requirements used for storing the graph. The alternative is to store the global state with the node and use global state equivalence. It is not necessary to store every global state; only those at the branch points (that is, the state upon entering the node).

The *PEW* builds a behaviour graph by building an execution tree breadth-first. The node-equivalence comparison depth D is specified by the user. Whenever a node is built which is the rightmost leaf of a subtree of height D rooted at some node N, then this subtree is recursively compared to depth D against all other subtrees of that height. If a match with a subtree rooted at some node M is found, then all arcs entering the node of greater depth have their endpoints changed to point to the node of smaller depth.

A typical method of building a tree is to recurse, and backtrack whenever a leaf is reached. In terms of the *PEW*, backtracking would mean that the entire state of the system would have to be stored at each node, so that it could return to this previous node and state and explore a new child from that point. This is both highly memory intensive, and would require substantial modifications to the interpreter.

A non-recursive alternative is to choose a node which has unexplored children (the *target* node), execute from the root to that node, and explore one of the children. Execution is then terminated, a new target node chosen, and the process repeated. This eliminates the need to store any extra information with the nodes and is easy to implement. It has the disadvantage of being more computationally intensive – for every node explored, execution must proceed from the root node to that node.

Once the *PEW* starts merging equivalent nodes it still uses the same basic algorithm; the algorithm is unaffected by cycles in the graph.

Definition 13 *A* candidate node *of an execution tree is a node with at least one unexplored child.*

Definition 14 *The* target node *of an execution is the leftmost candidate node with smallest depth.*

The graph building algorithm used by the *PEW* is listed in Algorithm 1.

The reference to Algorithm 2 is the point at which the node comparison takes place and the tree is converted to a graph.

The *PEW* thus performs a (computationally inefficient, but space efficient) breadth-first exploration of the execution tree.

```
    DOTARGET:
while there is a target node $T$
    $N$ = root node in preinitial state
    DONODE:
    Execute from start of $N$ to next branch point
        (that is, end of $N$), recording transition
        sequence in $S$.
    if $N$ has been explored before
        then begin
            check that $S$ is consistent with previous sequence[3].
            Set $N$ to be the next child
                en route to the target and resume
                at label DONODE
        end else begin
            Associate the sequence $S$ with node $N$
            if branch due to unreliable OUTPUT
                then create two child nodes
                else create child nodes for each DELAY
                    transition with all other clauses satisfied
            Mark the children as unexplored candidates
            if $N$ is the rightmost child of its parent
                then if $N$ has depth $\geq D$
                    then apply Algorithm 2
            Resume at label DOTARGET
        end if
end while
```

Algorithm 1: Behaviour Graph Building Algorithm

```
Let $M$ be the root node of the subtree of height $D$
    with rightmost leaf $N$
for $L$ = root nodes of all other subtrees of height $\geq D$
    from left to right in order of increasing depth
begin
    if $M$ and $L$ have identical sequences
        and $M$ and $L$ have identical subtrees of depth $D$
            then begin
                set all arcs that enter $M$ to have their
                    endpoints at $L$ instead
                    remove the subtree rooted at $M$
                    return
            end if
end for
```

Algorithm 2: Node Matching Algorithm

7.2.3 Practical Considerations

Clearly, the greater the cutoff depth used in the recursive definition of node-equivalence, the less chance there will be that the approximate definition will report two nodes as equivalent when in fact they are not. There must exist some optimal depth D such that:

- any two nodes M and N that are node-equivalent at depth D, are also node-equivalent at all depths $d > D$;
- there are no nodes M and N which are not node-equivalent at depth D but are node-equivalent at some depth $d > D$;
- there is at least one pair of nodes that is node-equivalent at depth D but not node-equivalent at depth $D - 1$.

There is no known *a priori* way of determing the optimal depth for comparison; the suggested method is to start with some small depth and increase it by one each time until two increases have gone by with no difference in the resulting behaviour graphs.

Causes should have their effects within the limits of node comparison. A typical cause-effect situation is a transition `OUTPUT`ing a message and some other transition dequeuing that message. The number of intermediate transitions that occur between these two events depends on the length of the channel queue, amongst other things. Here again it is clear that the semantics of IPC in Estelle can cause problems with an explosion in states if channel queues are allowed to grow. As a result, to use the method in practice, we modify the Estelle specification to use channel blocking.

This is easy to achieve using a second control channel for each channel which is used to signal that the channel is no longer blocked.

7.3 Using the Behaviour Graph

Once a behaviour graph has been built, it can be used to predict the performance of the specified system. Before a prediction can be done, a set of parameter values must be specified. These are the loss probabilities of each unreliable `OUTPUT` statement, and the delays associated with each `DELAY`ed transition. The behaviour graph itself is independent of these values, and of the probability distributions of the delays. At this stage, the *PEW* only allows constant delay values, not delay distributions. The method could, in principle, be extended to delay distributions, but the resulting parameterised graph would be much larger. Using constant delays the parameterised graph is often smaller than the behaviour graph itself.

Restriction 4 *All delay values are constants.*

Given a set of parameters, a *parameterised behaviour graph* is created from the behaviour graph by the graph processing program. This parameterised graph describes a semi-Markov process which can be solved and used to obtain the throughput of any subset of transitions in the graph. The *parameterised behaviour graph* is similar to a behaviour graph but includes delay and probability labels on each edge. The parameterised behaviour graph represents the possible execution traces that can occur in the system under the given parameter values.

7.3.1 Labelling the Graph

For arcs due to unreliable `OUTPUT`s, the labelling is trivial: the delay is zero and the probability is the probability of loss (right arc) or no loss (left arc).

Delays, on the other hand, add considerable complexity. When all the clauses other than the delay clause of a delay transition are enabled, the scheduler notes the time at which this first occurred. This time is the *trigger time* Υ of the transition. Two things can then happen:

- before the delay transition fires, some other transition t fires and causes one or more of the clauses of the delay transition to fail. In this case the trigger time for the transition becomes undefined, and we say the transition t *killed* the delay transition;
- the delay transition is selected for execution before any clauses are killed by other transitions. This will occur at time $\Upsilon + d$, where d is the delay associated with the transition.

In the case of the producer-consumer example, the producer transitions are each triggered both by themselves and the start of execution; this is because they have no clauses other than the delay clauses.

The decision of which arc to follow out of a node which ends at a delay transition scheduling point is based on the time at the node, and the trigger times and delay values of the various delay transitions at the start of each child node. The choice will be whichever arc results in the smallest defined $\Upsilon + d$ value.

Cycles in the graph make the labelling process more complex, as we can re-enter a node with a different set of trigger times. This problem requires that instead of storing the trigger times of `DELAY`ed transitions we store instead the time remaining to fire; that is, $\Upsilon + d - T$, where T is the current time.

Definition 15 *If N is a node in a behaviour graph with n outward arcs pointing to nodes $M_1, \ldots, M_n$, and each node M_i has a transition sequence which begins with a delay transition T_i, then the* history vector *of node N is the vector $(\tau_1, \ldots, \tau_n)$ where τ_i is the time remaining before T_i is due to fire, or undefined if T_i has some other clause which is not enabled.*

If the history vector of a node is known, it is a simple matter to determine which transition will be selected for execution.

The *PEW* places some restrictions on triggers and killers to simplify the labelling algorithm:

Restriction 5 *Trigger and killer transitions are unambiguous.*

This restriction states that if transition t_i is a trigger of t_j, then every time t_i is executed it must trigger t_j, and similarly for killers. Note that it is only delay transitions which require unabiguous trigger and killer transitions, and it is a trivial matter to modify a specification so that this is indeed the case.

The labelling process recurses depth-first through the graph, starting at the root node. Before recursing, data files containing the known triggers and killers for each `DELAY` transition are read, as well as the parameter values to be used for this labelling. While following a path, the current time is maintained. Whenever a trigger or killer transition fires, the time at which this occurs is also noted.

`OUTPUT` branches in the graph are labelled with the appropriate loss probability parameter values, and each path is then labelled separately with a recursive call to the labelling algorithm. If a node with child `DELAY` branches is entered, the most recent times for each child's triggers and killers are inspected. If a killer executed more recently than a trigger, the history vector entry for that child is undefined; otherwise it is set to be the current time less the trigger time plus the `DELAY` parameter value. The child transition with the smallest defined value in the history vector is chosen and the arc is labelled with this value; all the other outgoing arcs are set to have probability zero (this would not be the case if the restriction on constant delays was lifted). The labelling algorithm then follows this chosen path.

As the graph is labelled, a node may be re-entered but have a different history vector the second time. These nodes now represent distinct futures. When the new nodes children are (re)labelled, they too may have different histories, and are then also copied and relabelled.

```
labelNode(N) =
    if N ends with an unreliable OUTPUT transition
    then begin
        label left arc with probability of no loss
        label right arc with probability of loss
        labelNode(left child)
        labelNode(right child)
    end else begin
        work out the history vector of the node
        if the node is already labelled with a history vector
        then begin
            if there is a copy of the node with this history
                then we are done; return
            else copy this node and all children which end with unreliable OUTPUTs
        end
        choose the child with the shortest defined vector entry
        prune all other child arcs
        labelNode(selected child)
    end
end
```

Algorithm 3: Behaviour Graph labelling algorithm.

Recall that in the original behaviour graph each node was distinct. For each of these distinct nodes, a list of all copies made is maintained. When re-entering a node and before performing a copy, the list for the node is consulted; if there is already a copy with the new history, this is re-used and the recursive labelling bottoms out.

The labelling algorithm is summarised in Algorithm 3.

The parameterised behaviour graph for the producer-consumer, with the first producer transition having delay 7 and reliability 0.9 and the second producer transition having delay 2, is shown in Figure 7. Each arc in this figure is labelled with either a probability or a delay, depending on the branch type.

Consider how Figure 7 was obtained from Figure 6. Execution begins at the root node of the behaviour graph, with a history of $[7, 2]$. That is, transition $p1$ has a delay of 7 before it can fire, while transition $p2$ has a delay of 2. Transition $p2$ is chosen, and the right arc followed to the node containing $p2, c2$. This node now has a history of $[5, 2]$, as $p2$ must once again wait for 2 time units before it can fire, while $p1$ now has a remaining delay of 5. Once again, $p2$ is chosen; this cycles back to the same node, only this time the history is $[3, 2]$, so the node must be duplicated. Once again $p2$ is chosen, and once again there is a cycle back, this time with a history of $[1, 2]$; another duplication is done. Now $p1$ is chosen as it has the shortest delay, and we move to the node containing $p1$.

Figure 7: A Parameterised Behaviour Graph of the Producer-Consumer System

Note that the delay of the arc followed is one, as this is the remaining delay of $p1$. The new node containing $p1$ thus has a history of $[7, 1]$; that is, $p2$ has a remaining delay of 1. We continue in this fashion until a node is re-entered with an identical history vector. The path through the behaviour graph is then terminated. The behaviour graph with node histories shown on the right of each node is given in Figure 8.

The only branches that occur in parameterised graphs are loss branches, which are clearly stochastic. The graph can therefore describes a semi-Markov process. The imbedded process can be solved to obtain the proportion of executions of each node. The delays can then be introduced and the results adjusted accordingly. This is described in the next section.

When solving the graph, interest is typically focused on a single process, so transitions belonging to other processes can be ignored after the parameterisation has been done. This may result in 'empty' nodes, which can be removed. Any arcs going to them can be replaced by sets of arcs going to their children, with adjusted probabilities and delays (see Figure 9). The end result of this is often a considerably smaller graph. Figure 10 show the graph for the consumer process. This has been simplified even further by collapsing linear sequences of nodes into single nodes; the numbers in parentheses are the delays between the transitions.

7.3.2 Obtaining Throughput Measures

The result of solving the imbedded Markov chain described by the graph is the proportion of executions π_N spent in each node N. From these figures, throughput figures can be obtained directly from Equation 3:

$$Throughput(N) = \frac{\pi_N}{\sum_i(\pi_i \sum_j p_{i,j}\tau_{i,j})}$$

If a transition t occurs n_N times in node N, then the throughput of the transition is given by:

$$Throughput(t) = \sum_N n_N Throughput(N)$$

7.4 Discussion

While the behaviour graph technique is well-suited to dealing with finite-state machine based formalisms, Estelle has some characteristics that make it unsuitable. The main problem is the asynchronous communication between processes. This means that causal relationships between two transitions in separate processes may not be obvious.

A further problem is that by allowing any combination of delay values, we often admit many sequences that would never occur in the specification when it has a set of parameters. A simple example illustrates the problem well. This example applies to the AB protocol specification in Chapter 11, which had to be modified to prevent this situation arising.

Figure 8: Parameterised behaviour graph of the Producer-Consumer System showing Node Histories

Figure 9: Removing an Empty Node from the Graph (note how the probabilities are changed).

Figure 10: Parameterised Behaviour Graph of Consumer

Consider a timeout transition with a delay which is enabled when a message is transmitted. Asssume there is a provider delay associated with the transmission of the message. To examine the execution of the system under all possible sets of delays, the possibility that the timeout is shorter than the time required to send the message must be considered. Thus a time out may occur even before the message has been completely sent! Depending on the relative value of these delays, time outs may occur *any number of times* before the message is sent. If this happens, the interaction point queues become longer, and the throughput gradually degrades to zero in the limit. There are an infinite number of different execution traces in this situation, as there is no steady state. The only way to get around this problem is to use channel blocking in the specification.

A disadvantage of the behaviour graph method is the state-explosion problem. The behaviour graph method suffers from this problem less than other state-exploration techniques as it compares transition sequences rather than states. Different global states producing the same transition sequences are equivalent in the behaviour graph.

To reduce state-explosion, there are a number of techniques that can be used, including reducing the number of `DELAY` clauses and unreliable outputs, and using a loopback provider; that is, using a single protocol system which has its own interactions returned by the provider. This results in a substantial simplification of the size of the problem, and is the technique we have used in our examples (lacking the hardware to explore hundreds of thousands of states).

The use of a behaviour graph for performance prediction bears some similarity to the method proposed by Rudin ([Rud83, Rud85]). The primary differences between Rudin's method and the method used here are:

- Rudin's machine-readable formalism is never explicitly given; the *PEW* uses standard ISO Estelle;
- Rudin's method assumes each event takes a single time unit; the *PEW* can handle any (constant) delay values;
- It is not clear how Rudin's method can be used predictively; the behaviour graph method is completely oriented to prediction;
- Rudin's method assumes synchronous communication; the behaviour graph method simply requires that as the growth of channel queues be strictly controlled.

Part III

The Protocol Engineering Workbench

Chapter 8

The Protocol Engineering Workbench

The *PEW* is an integrated system containing an editor, an Estelle compiler, an interpreter, a debugger, and separate performance analysis and graph processing modules. The graph processor also assists in protocol testing. The *PEW* environment has been described previously in [KW90, Whe90a]. This chapter describes the structure and implementation of the *PEW*.

8.1 Historical Development

The *PEW* was designed as a tool for specifying and testing Estelle specifications, with simulation through a meta-implementation. The original intent of the simulation capabilities was for performance studies, as well as for obtaining model parameters for the closed multiclass queueing models proposed by Kritzinger. An early version of the *PEW* generated *MicroSNAP* code for such a model after executing a specification. However, it was difficult to determine to which servers transitions should be allocated. This problem led to the development of the simpler semi-Markov model described in Chapter 6, and the implementation of the *performance analyser* program, which is a tool for obtaining throughput figures from the stochastic matrix and holding time matrix of a semi-Markov process.

In investigating the use of the semi-Markov model for prediction, the *PEW* was enhanced to allow batch executions in which a *CONST* value (specifying a timeout, for example), could be varied over some linear range. Information about the effect that executing a transition had on clauses of other transitions was also collated, to assist in determining causal relationships between transitions. All of this was aimed at determining relationships between system parameters and the resulting sequences of transitions. This investigation led eventually to the development and implementation of the behaviour graph method, with the information about causal relationships being used to determine triggers and killers.

The *PEW* was originally developed under MS-DOS on an 80286 PC; a very modest architecture which had some implications for its design and implementation. When the behaviour graph method was implemented, the *PEW* was ported to Unix to allow larger systems to be analysed. The Unix version is a command-line version, while the DOS version has a much more sophisticated and powerful user-interface. It is the original DOS version which we describe in this chapter.

8.2 Structure of the *PEW*

The *PEW* consists of several components, of which the most important are illustrated in Figure 11. The components are shown as rectangular blocks, while the files of information input and output by the components are shown as ovals. The *PEW* has been implemented in Borland's Turbo C under the MS-DOS operating system, as well as on a Sun SPARCstations under the BSD4.1 operating system, and under Unix SVR4/386.[1] The source code consists of about 30 000 lines of C, excluding the Estelle-to-C translator system. The same source code is used on both DOS and Unix platforms; the differences are accounted for by the use of conditional compilation.

Starting at the top, we have an Estelle specification, created using the text *editor*. This editor includes basic syntax-directed editing features and language help. The specification is then compiled by the *Estelle compiler*, which produces a symbol table and a file of pseudo-code (which we call *E-code*) for use by the interpreter. The interpreter provides a meta-implementation of Estelle, and can be used to execute specifications in a controlled environment. Various information is output by the interpreter as a result of the execution. This includes a *transition sequence count* matrix and a *transition sequence delay* matrix. These matrices can be used as input to the *performance analyser* (after converting the transition sequence count matrix to a stochastic matrix). The performance analyser solves imbedded Markov chain models using Gaussian elimination.

The interpreter can also be used for exhaustive analysis of a protocol; this results in the construction of a *behaviour graph* which can be processed by the *graph processor* to produce throughput figures for different parametric configurations.

A few other small utilities are also part of the *PEW* (for example, a utility to convert the matrices output at the end of an execution by the *PEW* into an input file suitable for use by the performance analyser).

We now consider the various components in more detail.

[1]Turbo C is a trademark of the Borland corporation. MS-DOS is a trademark of the Microsoft corporation. Unix is a trademark of AT&T Laboratories. Sun and SPARC are trademarks of Sun Microsystems.

Figure 11: Structural Components of the Protocol Engineering Workbench

8.3 The Compiler

The *PEW* compiler supports most of the facilities of Estelle as defined in ISO IS 9074. The compiler is based on Brinch Hansen's Pascal compiler [BH85]. Recursive descent parsing is used to produce target code for an instruction set which we have designed for the meta-implementation of Estelle. This *E-code* is a superset of Hansen's version of Pascal's p-code. The target architecture is a hierarchical collection of communicating stack-based virtual machines, one for each executing process.

An advantage of the virtual machine approach is that we can make our E-code as simple or as complex as we like. While a large subset of the E-code instructions are simply primitive stack operations adopted from Pascal's p-code, a number of high-level instructions corresponding to Estelle-specific constructs have been added. For example, the E-code includes a `WHEN` instruction corresponding to a `WHEN` clause, and an `OUTPUT` instruction corresponding to an `OUTPUT` statement. The full set of E-code instructions is documented in [Whe90b].

The compiler can parse Estelle code directly from the in-memory doubly-linked list used by the editor, eliminating disk accesses with a resulting increase in efficiency. Compilation errors result in a return to the editor with the cursor positioned at the offending token.

Integration of the compiler and interpreter enables the interpreter to make use of the compiler's symbol table, eliminating the need for additional symbolic information for debugging purposes. The exceptions to this are source code line number information and block scoping information, both of which are represented as E-code instructions. This is more appropriate as they map positions in the E-code to positions in the source text and scope levels.

Some restrictions have been made to the Estelle standard to simplify the compiler. Although the standard allows partial specifications using unvalued constants, untyped variables and external procedures and functions, the *PEW* compiler accepts only complete specifications. Packed data types and real-valued data types are not supported, nor is the `ALL` clause. The variant forms of `NEW` and `DISPOSE` are also not supported. Conversely, a few extensions have also been made. As the *PEW* essentially provides a simulation environment, Pascal-like I/O procedures were added, as well as a number of other useful functions and procedures. Examples are a function to return the simulation time, and one to return the length of message queues. Pseudo-random number functions supporting several different probability distributions have also been added.

Of particular interest are a number of compiler directives to control aspects of the execution. These include:

- specifying the reliability of message passing commands (`OUTPUT` statements). This is useful for testing systems where the communication channel is unreliable;
- specifying the probability distributions of `DELAY` clauses;

- specifying whether the output from `WRITE` and `WRITELN` statements is to be directed at files, the screen, neither or both. Disabling both speeds up execution considerably;
- overriding the FIFO nature of interaction queues;
- specifying the beginning and end of transition groups.

The transition group directive causes matrices to be generated for a subset of the transitions only. This is one of the strategies used for helping to determine mapping functions for the semi-Markov process models.

The compiler operates in a single pass, producing special instructions when forward references occur. A three-pass linker is used to resolve these references and remove the (now redundant) special instructions. The linker also performs peephole optimisation of the E-code. The code is output to a buffer in memory, from where it is read by the interpreter.

Stand-alone versions of each of the components of the *PEW* can also be generated. These do not perform quite as well (due to the use of disk storage rather than memory for source and target code), but allow larger specifications to be compiled and executed under MS-DOS.

8.4 The Interpreter

The interpreter consists primarily of two components: a *scheduler*, and an *E-code virtual machine* (VM). We call this VM the E-machine. It is an extension of the p-machine often used as a target by Pascal compilers.

An E-machine represents a single executing instance of an Estelle module. The complete executing system is a hierachy (tree structure) of E-machines. Each E-machine is represented by a number of components, including:

- the module header and body names;
- the module's class;
- the machine's *transition table*, which contains all known information about each transition (including several tables of information collected by the interpreter about the execution of the transition);
- a table of child machines;
- a pointer to the machine's parent, and the index number of the machine in the parent's child machine table;
- a memory area which is used as a stack and heap. Heap allocation is made from the end of the area while the stack grows from the start of the area. If no allocation is done from the heap,

then the stack can grow dynamically; once a heap allocation is done the memory area becomes fixed in size. The stack is referenced via a stack pointer and activation record pointer;

- the current state;
- the current instruction address;
- an *interaction point table*, which contains information about all the machine's interaction points, including the queue, length of queue, statistics gathered about the queue, debugging information and so on;
- a table of scope levels for symbol table access;

Some of this information in obtained when certain E-code instructions and interpreter functions are executed for the first time, some is gathered during execution, some is used as part of the execution (for example, the IP channel queues), and some is used to control the execution (for example, breakpoint information).

Ignoring the gathering of statistics and the symbolic debugging aspects, the core of an E-machine is the memory area, the channel queues in the IP table, the transition addresses in the transition table, and three registers **s** (stack pointer), **b** (activation record base pointer) and **p** (instruction pointer). These are the same registers as used in a Pascal p-machine.

The E-code-specific instructions are generally much more complex than p-code instructions, which are mostly simple stack operations. The E-machine implementation uses a table-driven instruction dispatcher, using the E-code opcode as an index into a table of pointers to functions, one function corresponding to each unique E-code instruction.

Here is a typical p-code instruction function from the interpreter:

```
#define TOS stack[s]
#define PUSH(v) stack[++s] = v
#define POP() stack[s--]
#define PARAM() (short)code[++p]

void Proccall()
{
        short level = PARAM(), displ = PARAM(), returnLength = PARAM();
        short x = b;
        while (level--) x = stack[x];
        s += returnLength;
        PUSH(x); PUSH(b); PUSH(p);
        b = s - 2;
        p += displ-4;
}
```

This is the instruction for calling procedures or functions. It has three operands: the number of activation record links to traverse to get the appropriate dynamic activation record entry, the displacement in the E-code to the address of the code for the procedure, and the number of words required to save the returned value. This number of words is reserved on the stack, a new activation record is created, and the instruction pointer is modified to point to the function or procedure.

The following example is for the E-code instruction `When`, which is generated by the compiler for each `WHEN` clause. This instruction expects an index into the machine's IP table on the stack, and leaves this index value plus one on the stack if it is successful, or zero if it fails. This is one of the simpler E-code-specific instructions.

```
void When() {
        short ident = PARAM(), len = PARAM();
        short index = POP();
        IP *ip_ptr = VM->IPTable[index];
        INTERACTION *head = ip_ptr->queue->first;
        if (head && head->ident == ident && head->destIP == ip_ptr
                && head->time<=globaltime)
                        PUSH(offset+1);
        else PUSH(0);
        p+=3;
}
```

Each virtual machine is controlled by two functions, `evalClauses` (Algorithm 4) and `execBlock` (Algorithm 5). The `evalClauses` algorithm is called recursively for the entire tree of virtual machines, and records information about what transitions are immediately fireable, and which are enabled but have `DELAY` clauses. If the first list is non-empty, the scheduler selects a set of transitions to fire and then executes them by calling `execBlock` for each one in turn (this is done by a function called `executeTransitions`). If there are no fireable transitions, the scheduler chooses the transition with the shortest delay, updates the clock accordingly, and reevaluates the clauses.

Points to note from the `evalClauses` algorithm are:

- a channel-blocking option is supported by the *PEW*. This will prevent a transition from firing if there is already an interaction on the `OUTPUT` queue. If a transition has more than one `OUTPUT` statement, only the first is checked;
- the `DELAY` and `PROVIDED` clauses are executed within the scope of the `WHEN` clause, which introduces new identifiers for each of the interaction arguments (if any). For this reason an activation record is created before these clauses are evaluated and removed afterward;

```
loop through each transition in the table
    if first time this transition is being examined
        then record static info in table entry
    enabled = TRUE
    waiting = FALSE
    if using channel blocking and transition has an
        OUTPUT statement
        then if an interaction is queued at the endpoint
            then enabled = FALSE
    if transition has FROM clause
        then if the clause is not satisfied
            then enabled = FALSE
    if transition has WHEN clause
        then if it is satisfied[2]
            then push interaction arguments onto stack
            else enabled = FALSE
    make an activation record
    if transition has DELAY clause
        then if building behaviour graph or clause fails
            then begin
                waiting = TRUE
                enabled = FALSE
            end
    if transition has PROV clause
        then if it is not satisfied
            then begin
                waiting = FALSE
                enabled = FALSE
            end
    pop activation record
    pop interaction arguments from stack
    if enabled then record a fireable transition
    else if waiting
        then record an enabled DELAY transition
end loop
```

Algorithm 4: The evalClauses Algorithm

```
while opcode not ENDCLAUSE or ENDTRANS do begin
    if opcode is CONSTANT
        then push operand and increment p
    else if opcode is NEWLINE
        then begin
            synchronise display
            check breakpoints
        end
    else if there is a table entry for opcode
        then call the function pointed to by table
    else report illegal instruction
end while
```

Algorithm 5: The `execBlk` Algorithm

- if building a behaviour graph, `DELAY` clauses always fail;
- to determine if a `WHEN`, `DELAY` or `PROVIDED` clause is satisfied, `evalClauses` calls `execBlock`.

The `execBlock` algorithm is straightforward: it executes instructions repeatedly by calling the function pointed to by the function table entry corresponding to the current opcode. If there is no corresponding table entry, an illegal instruction is reported. Execution continues until an `ENDCLAUSE` or `ENDTRANS` opcode is encountered; the former indicates the end of code for a `WHEN`, `DELAY` or `PROVIDED` clause, while the latter indicates the end of a transition body. Two opcodes are handled explicitly in `execBlock`. Execution profiling shows that more than 50executed by the interpreter are `CONSTANT` instructions; these are thus handled inline to avoid the overhead of a function call. The `NEWLINE` instruction is used to synchronise E-code instructions with line numbers in the Estelle specification so that the process browser (described below) can position the cursor at the Estelle statement which is being executed.

The *PEW*s scheduler algorithm is shown in Algorithm 6. The algorithm loops repeatedly until the user quits or deadlock is detected. Other exit conditions are when a user-specified timelimit is exceeded, or the `pathAbort` flag is set. The latter flag is set by the behaviour graph algorithm to terminate exploration of a path. The scheduler calls `executeTransitions` (Algorithm 7) to select and execute a set of transitions.

As the *PEW* compiler and interpreter are integrated into a single system, the compiler does not destroy the symbol table it creates when compilation is completed, but instead passes this symbol table to the interpreter. This information is used to assist in implementing the *process browser* user interface to the interpreter. The process browser is documented in detail in [Whe90c], and is described in brief in Section 8.5.

```
while user hasn't quit and no deadlock do begin
    recursively call evalClauses
    if no fireable transitions then begin
        if there are enabled DELAY transitions
            then if building a behaviour graph
                then choose appropriate DELAY branch
            else update time by smallest remaining DELAY
        else report deadlock and exit
    else executeTransitions
    if timelimit exceeded then exit loop
    if pathAbort then exit loop
    if in interactive mode then get command
end while
```

Algorithm 6: The Scheduler Algorithm

```
doneSomething = FALSE
Count = 0
if building a behaviour graph
    then if there is a delay target in this VM
        then select it for execution
loop through all transitions in VM
    if transition is fireable
        then begin
            increment Count
            if highest priority so far
            then select transition for execution
        end
end loop
if Count is not zero but nothing is selected
    then select a fireable transition randomly
if a transition is selected
    then call execBlock to do it
    else loop through all child VMs
        call executeTransitions for child
        if a child transition was executed
            then if VM class is ACTIVITY or SYSTEMACTIVITY
                then exit loop
    end loop
```

Algorithm 7: The exececuteTransitions Algorithm

Must paste a figure in here

Figure 12: Process Browser Display

When the interpreter exits, it writes a file containing statistics and information about the execution. These include the enablement and firing of transitions, amount of traffic over channels and mean delays of events. The user may request various levels of detail in this execution record, up to a full execution trace. The transition sequence matrices are also included in this information file.

8.5 The Process Browser

The process browser is a multi-windowed user interface to the *PEW*s interpreter (Figure 12). At any stage the process browser shows information about one particular module instance (VM). The bottom line of the display indicates the name, type and state of the currently viewed VM, and the execution time. Four windows are displayed on the screen:

- the *source window* shows the Estelle source code;
- the *child window* shows the module variables of the VM;
- the *transition window* shows information about the VM's transitions;
- the *interaction point* window shows the IP variables and the channel queue contents of the VM.

The sizes of the windows can be changed by the user.

One window is always active, and has a moveable cursor. By pressing `ENTER` when the cursor is on a module variable entry in the child window, that child VM will be selected for display. A function key is used to select the parent of the current VM. It is thus easy to maneuvre through the VM tree. Pressing ENTER in any of the other three windows will cause a menu to pop up with commands applicable to the window. Each of these menus includes commands to set, clear and modify breakpoints on the associated item (line number, transition, or interaction point). The actions associated with breakpoints include returning to user control, adding symbolic information to the log file, and activating other breakpoints. Other menu commands allow statistical information to be viewed, jumping to specific source lines, showing connection endpoints of IPs, and so on. An interaction can be deleted from a channel queue by positioning the cursor on it and pressing the `DEL` key.

The process browser can be switched between an Estelle view and an E-code view of the specification; single-stepping is performed on the current level. In other words, when viewing E-code, a 'single-step statement' instruction single-steps on a single E-code instruction rather than on an Estelle statement. When viewing E-code, the interaction point window is changed to show the process stack.

An I/O window can be called up showing all I/O that occurs between the user and the executing specification. `WRITE` and `WRITELN` statements can be separately enabled and disabled, and their results targetted to either or both the I/O window and execution log file.

Execution can be controlled in a number of ways. A time limit can be specified, or single-stepping can be done at various levels (Estelle statements, transitions, scheduler iterations).

When building behaviour graphs the process browser screen is not updated completely. All that is displayed is the number of nodes explored and the amount of free memory available.

8.5.1 Time and the Scheduler in the *PEW*

The semantics of time in Estelle were described in Chapter 4. The only constraint that has to be satisfied is that the delays associated with transitions change uniformly with respect to one another. How best should time be implemented in an interpretive environment such as the *PEW*?

One approach would be to assume that every transition which does *not* have a `DELAY` clause takes one time unit to execute. This approach is clumsy; it forces us to scale all time units by an atomic quantity, which is the shortest delay associated with any transition. There may be transitions that are virtually instantaneous; by specifying that they have a delay of 1 we force the delay values of other more time-consuming transitions to be large numbers. A practical consideration is that by having large delay values, it may not take long before we have overflows occurring in variables used to store time values.

An alternative approach is to assume that every transition without a `DELAY` clause takes negligible time, which we treat as zero. This may seem artificial, as no transition is really going to take zero time; however, it is a more practical alternative to the first. It allows us to associate delays only with those transitions whose execution requires a relatively significant amount of time; furthermore, the atomic unit is the smallest of these *significant* delays. This approach allows us to model the other approach by giving every transition an explicit `DELAY` clause, while the converse is not possible.

To illustrate this more clearly, say we have three transitions, the first of which takes 50 microseconds, the second 50 milliseconds, and the third 1 second. If we use the first scheme, we will have to associate delays of 1000 and 20000 with the second and third transitions. With the second scheme, we ignore the time used by the first transition, and use delays of 1 for the second and 20 for the third. Obviously, the second scheme will take a thousand times longer before numerical overflow would occur on our clock counter variable.

Of course, we would not use the second scheme with no delay on the first transition if that transition was to execute a million times to every execution of the second and third transitions. However, this situation will rarely occur in practice (and probably indicates something very wrong with the protocol specification). We have opted for the second scheme in the *PEW*. In particular, the scheduling mechanism in the *PEW* has the following behaviour:

1. Examine all processes, and select a set of non-delayed transitions for execution. Execute these, and return to step 1. If there are no such transitions, go to step 2.

2. Examine all delayed transitions, and choose the enabled delay transition which has the smallest remaining delay. Update the clock by the remaining delay, shorten the remaining delays of all other delayed transitions appropriately, and execute the transition. Go back to step 1.

Thus, time only advances in the *PEW* when a delayed transition is executed. As long as there are non-delayed transitions to execute, time will not advance. This violates the first of the two assumptions about time in Estelle, but satisfies the important constraint that time moves at the same rate for all delayed transitions. We have found this to be a most satisfactory implementation for an Estelle interpreter. We regard the phrase ‘time does not advance’ as really meaning ‘the advance of time is imperceptible on the scale chosen’.

When executing transitions, the scheduler starts with the root process, and recurses depth first through the children. As all processes’ enabling clauses are checked before any transitions are executed, the fact that the transitions are executed in a sequence and not truly in parallel is not a problem. `WHEN` clauses will still be satisfied, and the parent-child priority relationship in Estelle guarantees that changes to shared variables will not cause problems with `PROVIDED` clauses.

8.6 Checking a Specification

The *PEW* performs some checks on a protocol when it compiles the specification. While this is far from a full verification, it assists in finding obvious errors. Amongst other checks, the following errors are reported:

- Interactions that are `OUTPUT` but never referenced in `WHEN` clauses;
- Transitions with `WHEN` clauses that refer to interactions that are never `OUTPUT`;
- States that can never be reached;
- States that can never be left.

The *PEW* also detects and reports global deadlock if it occurs when executing a specification. The behaviour graph technique can detect whether livelock or deadlock situations exist; an example of this is shown in Chapter 11.

Chapter 9

Performance Evaluation by Execution

The most general way in which the *PEW* can be used to evaluate the performance of a system specified in Estelle is by executing the specification. Unlike the analytical techniques, execution is unrestricted and can handle any valid, complete specification (subject to the restrictions of the compiler).

Before a specification can be executed, it must be made 'complete' by including appropriate user and provider layer processes. We describe the requirements of these processes and the facilities in the *PEW* which aid in their implementation, as well as the types of output that can be produced by the *PEW*.

9.1 System Structure

In order to execute a system specified in Estelle, it is necessary to add upper and lower layer processes representing respectively the *user* of the system's services, and the *provider* of services. For example, a link layer protocol specification based on the OSI model could have a provider process representing the physical layer, and a user process representing the network and higher layers. These upper and lower processes need not be complete protocol systems in themselves; they are just required to drive the system being studied, and represent a workload and operating environment. The typical structure of the resulting specification is illustrated in Figure 13.

In principle it should be possible to generate the user and provider processes automatically given a system to be studied, and the characteristic features to be provided by the corresponding user and provider (in fact this has been done in the EWS system described in the next chapter). The current implementation of the *PEW* requires these processes to be created by hand, but a future version of

Figure 13: Structure of an Executable Specification, consisting of User, Protocol and Provider Entities.

the *PEW* should overcome this.

9.1.1 The User Process

The user process is required to drive the system under study. A typical user process will make service requests of the system, and consume any responses. Some of these responses may in turn require further service requests. The complexity of the user process is in general dependent on the number of different service request and response primitives provided by the underlying system.

9.1.2 The Provider Process

The provider process is required to provide the implicit peer-to-peer communication between the systems being studied. The complexity of this layer is dependent on the number of 'features' of lower layers being modelled. As the provider process includes the physical layer, these may be numerous, and include:

- message loss and corruption;
- rearrangement of the order of messages;
- transmission and propagation delays;
- expedited message transmission.

Unfortunately, Estelle is unwieldy in implementing some of these facilities, particularly message rearrangement and expedited delivery. While it is straightforward in Estelle to associate a delay with the delivery of a message by the provider, allowing *separate* transmission and propagation delays is less so. The *PEW* provides a number of facilities for simplifying the simulation of these characteristics.

The provider process can pass interactions back to the same process it receives them from (much like the 'loopback' modes supported by communication hardware for testing purposes), or to a single separate similar process (for point-to-point communication), or to multiple processes (for broadcast communication). The latter is unwieldy to specify in the *PEW*, as the `ALL` clause is not supported. As a result, we have restricted our use of the *PEW* to point-to-point based protocols, in normal and loopback modes.

9.2 Facilities provided by the *PEW*

Several facilities have been incorporated in the *PEW* to make the implementation of user and provider processes easier. These include the specification of channel reliabilities and the implementation of several probability distributions for delays. These features are all enabled by using compiler directives embedded within comments in the Estelle specification. The use of these optional facilities thus has no effect on the Estelle specification itself, which can remain standard Estelle.

9.2.1 Channel Losses

The `$R`<*reliability percentage*> directive is used to specify the reliability of the next `OUTPUT` statement that follows in the text of the specification. For example, {`$R100`} specifies that the next `OUTPUT` statement in the specification is completely reliable, and no messages will be lost (this is the default). The directive applies to a single `OUTPUT` statement only; after compiling any `OUTPUT` statement the compiler sets the reliability back to 100%. When the *PEW* executes an `OUTPUT` statement, it generates a random number in the closed range [1,100]. If this number is greater than the reliability percentage of the `OUTPUT` statement, the message will be discarded rather than appended to the destination interaction point queue.

This directive is usually used only in the provider process, as it is this process which is responsible for modelling the physical layer, where such losses can occur. Furthermore, it is usually only used when the system being studied includes the link layer functions. On the other hand, if we were studying a transport level protocol, we could assume that the provider process includes a link layer which provides reliable communication, and we thus would not want the provider process to lose interactions.

9.2.2 Resequencing of Interactions

The `$QR` compiler directive is used to override the FIFO nature of message queues in Estelle. When this directive is used, the next interaction point that is declared has a random queueing order for all messages arriving at that point. Messages `OUTPUT` through the point are not affected. For both sides of a connection to have random queueing, both connection end points must have their declarations immediately preceded by the directive. This facility is useful for modelling the possible effects of dynamic routing in packet-switching networks, for example, in which two consecutive packets may take different routes through the network, with the first packet arriving later than the second.

Neither random queueing nor priority queueing (discussed next) will place an interaction at the head of the queue unless the queue is empty. This ensures that executable transitions with `WHEN` clauses are guaranteed the integrity of the message at the head of the queue.

9.2.3 Priority Queueing

The `$QP` directive is used to select priority queueing on a channel type. Channel declarations include the declaration of all the interaction types that may pass through that channel in either direction. The *PEW* allocates priorities to the interaction types (separately for each direction or role), starting with zero and increasing by one each time. Zero is the highest priority. In order to declare priority *classes* (that is, groups of interactions with the same priority), the priority counter can be set to a specific value at any point with the `$PR=`<*value*> directive. An example of a priority channel declaration is:

```
{$QP Select priority queueing}

CHANNEL prot_prov_chan(prot_role, prov_role);
    BY prot_role, prov_role:
        {** REJ frames have highest priority **}
        REJ(send_seq:seq_type; recv_seq:seq_type);
        {** RR frames have priority 1 **}
        RR(send_seq:seq_type; recv_seq:seq_type);
        {** RNR has same priority as RR **}
        {$PR=1} RNR(send_seq:seq_type; recv_seq:seq_type);
        { I frames have priority 2 which is the lowest }
        I(send_seq:seq_type; recv_seq:seq_type; data:data_type; ok:BOOLEAN);
```

When the *PEW* `OUTPUT`s an interaction over a priority channel, the interaction gets queued after the first interaction already in the queue having the same or higher priority. It thus moves ahead of any lower priority interactions at the tail of the queue. The only exception is that an interaction already at the head of the queue will not be displaced.

9.2.4 Corruption of Interactions

The *PEW* has no inbuilt facility for message corruption. This is achieved more easily by including protocol-specific code in the provider process. For example, if a CRC check is used for detecting corrupted interactions, the provider could randomly change the CRC field of an interaction before passing it on. A simple method is to include a Boolean flag which indicates whether the containing frame is corrupt; this can then be set randomly with a certain probability. The I-frame declaration in the priority queue example above contains such a field. An alternative that is assumed when building behaviour graphs is that interaction corruption could be considered the same as loss.

9.2.5 Delay Distributions

The Estelle standard does not specify how time is distributed in `DELAY` clauses. This is because `DELAY` clauses are used to specify the value of timeouts and other time values in a specification, not to specify the arrival rate of service requests or provider responses. However, when using the *PEW* for performance evaluation, `DELAY` clauses are used in the latter manner in the provider and user processes. These delays are typically not uniformly distributed. The *PEW* thus provides a selection of distribution types that can be used in `DELAY` clauses, namely exponential (`$E`), Poisson (`$P`), Geometric (`$G`) and uniform (the default).

Although these distributions generate values up to infinity. `DELAY` clauses in Estelle have a specified minimum and maximum value. These minimum and maximum values can still be used; the *PEW* will then generate delays in the subrange specified by these values. If a delay value outside of the range is generated, it will be discarded, and a new one generated, until a value in the range is generated. In most cases, we can limit the maximum value to a fairly small value without seriously affecting the distribution, as the probability of a delay being greater than the maximum is sufficiently small as to be negligible. It should be noted however, that the delay values generated in the *PEW* are discrete integers, whereas the real distributions are continuous (except for the geometric distribution). Thus the *PEW* can only approximate these distributions at best.

The compiler directives to specify these delay distributions require a distribution type specifier, and an expected value. The latter is the mean of the distribution. For example, the `DELAY` clause:

```
{$E10} DELAY (1,*)
```

specifies an approximate exponential distribution with expected value of 10, but a minimum value of 1 and maximum value of 32767.

The *PEW* also supports random number generating functions which use the same routines as the `DELAY` clause. These take three arguments, representing the mean value, and minimum and maximum values as in the `DELAY` clause. The numbers generated by these routines can be used in `DELAY` clauses as well to achieve the same effect as a delay distribution. The use of the routines has the disadvantage that the specification itself becomes non-standard (as the routines are *PEW*-specific), but have the advantage that they can be modified in the parametric execution method described in section 9.6, whereas in the current implementation the compiler directives cannot.

9.2.6 Propogation Delays

The use of `DELAY` clauses in the provider process allows the modelling of end-to-end delays in a system. Sometimes this is not sufficient. Consider a packet switching network with a sliding window protocol. There is a limit to how fast the packets can be sent by one side (the transmission time) as well as a (possibly constant) propagation delay for the packets to reach the other side. The fact that the propagation delay is often significantly longer than the transmission time was one of the reasons for the development of sliding window protocols. Another example in which this occurs is in microwave transmission.

Simulating the separation of these two delays in Estelle can lead to a considerably more complex provider layer. The provider needs to have delayed transitions in which packets are dequeued from the system (the delay here being the transmission time) and then buffer these packets internally until the propagation delay has elapsed, before the packets can be `OUTPUT` to the peer system. The best solution in standard Estelle is to include an internal communication channel within the provider, which is used to obtain a two-stage delay - one delayed transition dequeues the packet from the system and `OUTPUT`s it to the internal channel, while another delayed transition dequeues it from the internal channel and `OUTPUT`s it to the peer.

The *PEW* provides a `$D=`<*delay*> directive to address this problem. This directive is used before an interaction point declaration to associate a propagation delay with interactions `OUTPUT` through that interaction point. The *PEW* timestamps each interaction that is `OUTPUT` through the interaction point with its arrival time, and these arrival times are checked by the *PEW* when evaluating `WHEN` clauses. If the interaction at the head of a queue has a timestamp later than the current time, all `WHEN` clauses referring to that interaction point will fail.

The current version of the *PEW* only supports constant propagation delays, and does not allow the simultaneous use of propagation delays and random or priority queueing on a single connection. We have not used this facility in our case studies as it cannot be used in conjunction with the behaviour graph method.

9.3 Understanding the Output

The *PEW* produces an extensive listing summarising the results of an execution. These results are produced automatically at the end of an execution, and can also be produced as a result of the execution of breakpoints in the specification by the debugger, or explicitly by the user during execution. An excerpt from such a listing is shown in Figure 14. This is from a simple two-process

Figure 14: Sample Excerpt of Execution Summary Statistics

Figure 14: Sample Excerpt of Execution Summary Statistics *(continued)*

producer-consumer system. The `p_mod` process is a producer with two transitions, each `OUTPUT`ting different messages at different intervals. One of the two `OUTPUT`s is unreliable. The consumer `c_mod` simply consumes these messages using two transitions with appropriate `WHEN` clauses. The full specification of this system is given in Figure 4, Chapter 7.

Shown in this figure are:

- the *transition sequence count table* showing how often each transition in a process was followed by others. This information is used in to derive the semi-Markov process models. Transitions can be grouped together, with the sequence counts built separately for each transition group. This is useful to restrict attention to a particular subset of transitions. An entry in the i^{th} row and j^{th} column indicates that the i^{th} transition was followed consecutively by the j^{th} transition that many times. In the example it can be seen that the second producer transition was followed directly by itself 3570 times;
- the execution summary for each transition, showing how often it was enabled and fired, the time it first fired, the time it most recently fired, the mean delay between it firing and the prior transition firing, the mean delay between occurrences of itself (the *cycle* or *recurrence* time), and the mean delay between it firing and the next transition firing. For example, it can be clearly seen that the first producer transition output messages at an average rate of one every seven time units, while the second did so every two time units;
- A summary of the traffic dequeued through each process' interaction points. This includes the total number of dequeued interactions, the maximum length reached by the queue, the mean time spent by dequeued interactions in the queue (that is, the mean time between an interaction being `OUTPUT` and being dequeued by a `WHEN` clause), and the maximum time spent in the queue by any dequeued interaction;
- A summary of the interactions sent by each process through each of its interaction points,

Figure 14: Sample Excerpt of Execution Summary Statistics *(continued)*

including how many were sent successfully and how many were lost (the latter only occurs if the reliability of an associated `OUTPUT` is less than 100%);

- The contents of the queues associated with each process' interaction points at the time of termination, including the length of the queue, and the time at which the interactions were `OUTPUT`. Only the first few interactions at the head of the queue are shown.

At the end of the listing are two additional tables produced by the *PEW*. The *global transition count table* shows the transition sequence counts for all processes, as well as the *linkages* between layers. Note that the diagonal submatrices of this table correspond to the transition sequence count tables for the individual processes. The entries outside of these diagonals are the linkage counts. A linkage count entry of n in the i^{th} row and j^{th} column indicates that that n interactions that were dequeued by transition j were output by transition i. For example, transition 1 (the first transition in the consumer) dequeued 1265 interactions that were output by transition 3 (the first transition in the producer). The *global transition sequence delay table* shows the mean delays that were measured between each consecutive pair of transitions. Usually only the diagonal submatrices will have non-zero entries, unless a global matrix for the whole system is requested by the user. The sequence delay in row i column j is measured from the time that transition i began execution to the time that transition j began execution. If a transition which has a delay is executed, the transition only executes after that delay has elapsed. Thus the measured delay is more dependent on the delay associated with transition j than transition i.

In terms of measuring performance, the most useful information is the execution counts for the various transitions, which can be used to determine performance measures such as throughput. The information in the tables can also indicate problems in a protocol (for example, a queue which is becoming increasingly long due to the arrival rate of interactions exceeding the service rate of these interactions), possible redundancies (transitions that never execute), and other characteristics.

9.4 Interaction Time Traces

Another useful facility of the *PEW* is the ability to generate *time traces* of the exchange of interactions. Such a trace represents a graph of the exchange of interactions in which the one axis is the scheduler iteration (and hence time), and the other axis represents the different processes. Each time an interaction is output, the interaction name is printed at the appropriate place in the trace. These traces have been useful in quick checks of protocols to determine if they are behaving obviously incorrectly. Figure 15 shows an example of part of such a trace.

Figure 15: Part of an Interaction Time Trace

Figure 16: Part of a Clause Effect Trace

9.5 Clause Analysis

The *PEW* can also generate a file during execution which gives information about the effect that executing each transition had on the clauses of other transitions. An excerpt from such a file is shown in Figure 16. This example shows the information for two transitions.

Each entry consists of a row of information about the executed transition, followed by zero or more rows of information about the affected transitions. These latter rows are each indented. The information given for the executed transition is:

- the *meta-index* of the transition;
- the time at which it fired;
- the name of the module variable to which the transition's containing VM is bound;
- the index of the transition in the VM's transition table.

Thus the first row indicates that the second transition of the process bound to the `ua` module variable was executed at time 1. This transition has meta-index 30. The meta-index is a unique integer identifying a particular transition in a particular module instance. These correspond to the rows and columns of the global transition sequence matrices.

The information given for each affected transition is:

- the meta-index of the transition;
- the difference in clause states;
- the clauses that were satisfied and remain so;
- the old and new clause states;
- the module variable and transition index.

Thus for example, the execution of transition 30 at time 1 resulted in the `WHEN` clause of transition 46 becoming enabled, while the `PROVIDED` and `FROM` clauses were already enabled and were unaffected.

This information can be useful in determining mapping functions for the semi-Markov models. A utility has been implemented which processes the information in this file and produces a summary of the effects of the execution of each transition. The information produced is the number of times each transition executed, as well as the number of times the execution had each of the following four possible 'effects' on each clause of each transition in the system:

- clause was not satisfied but became satisfied;
- clause was satisfied but became not satisfied;
- clause was satisfied and was unaffected;
- clause was not satisfied and was unaffected;

The number of times of each of the four cases are also given as percentages.

Although the *PEW* made use of some of this information at an early stage (in the behaviour graph method), outputting and summarising this information is a recent addition to the *PEW*. Early impressions are that this information should be extremely useful although exactly how to use it is still being considered.

9.6 Parametric Executions

A primary application of performance evaluation is fine-tuning a system for optimal throughput. This involves modifying system parameters such as timeout delays, buffer sizes, and so on, to determine the best combination of values. In some cases, there may be a trade-off between performance and resource requirements. For example, memory is needed for buffers; a buffer cannot be increased in size if there is no further memory available.

The execution capabilities of the *PEW* can be used for such studies. Parameters can be changed, the specification recompiled, further executions performed, and the effect on the throughput noted. Often this process is tedious, requiring human intervention each time a parameter is changed. For parametric studies where the independent variable can be represented by a single `CONSTANT` in the specification, the *PEW* can automate this process. It would be straightforward to extend this to the case of more than one variable.

This is achieved using the `Analyze` menu of the *PEW*. This menu allows the user to specify the independent variable, the minimum and maximum values it should take, the increment to be used between each execution, and the length of execution to be used each time. The *PEW* recompiles the specification for each value in the specified range, executes the specification for the appropriate length of time, and tabulates the results.

The results thus produced are different to the execution log that was described in the previous section. One or two tables are produced, the first tabulating the execution counts of each *named*

Figure 17: Sample Parametric Execution Results

transition in the specification. This naming of transitions is a standard feature of Estelle; the `NAME` directive is one of the clause types in the language. It has no semantic significance in Estelle other than for reference purposes, but has been used in the *PEW* to identify important transitions. The execution counts of a named transition are averaged over all processes that contain that transition.

The second table tabulates the results of user-specified expressions involving the named transitions and a special variable called `time`, which represents the execution time specified by the user. For example, if there is a transition to receive an acknowledgement for a correctly transmitted interaction, the execution count of this transition is the number of interactions that were correctly transmitted during execution. If this transition has the name `GetACK`, we can specify the following expression in the `Analyze` menu to evaluate the mean throughput of all the protocol processes in the system in terms of interactions sent per unit time:

```
throughput = GetACK / time
```

Up to eight such expressions can be defined. Figure 17 shows the results produced when varying the timeout of an AB protocol specification, and computing the throughput as described above.

Chapter 10

Other Protocol Engineering Tools

The problems of protocol engineering have led to the development of several Computer-Aided Software Engineering (*CASE*) tools for protocol development and testing. In this chapter we describe some other Estelle-based tools, all of which were developed as part of the European Strategic Programme for Research in Information Technology (*ESPRIT*)[DAC$^+$]. The objectives of this project were to produce a set of tool prototypes to assist in formal description, validation and implementation of protocols within an open, extensible environment. The tools are aimed at testing, implementation and simulation; none of them use an analytical approach like that in the *PEW*, which provides considerably more powerful performance evaluation capabilities.

There are many tools aimed at performance evaluation that use other specification methods (see for example [SMC90, GR91, MIL89, JMG88, 12590, Fle88, Hol91, New91]); however, protocol-oriented performance evaluation tools are rare and almost always use simulation to achieve their task. Analytical tools are usually either oriented at specific classes of protocols (in particular LANs and network layer routing protocols) or use model-oriented rather than implementation-oriented specification methods. The *PEW* is thus unique in being a (mostly) general-purpose implementation-oriented tool which allows three different approaches to performance evaluation (simulation, analytical model generation, and behaviour graphs).

10.1 *ESTIM*

Estim (Estelle Simulator based on an Interpretive Machine) [JdSS92] is a tool that allows the simulation of Estelle* specifications. Estelle* is a version of Estelle that supports a rendezvous mechanism for synchronous communication, which in turn allows efficient process algebra verification techniques to be used (as well as helping to ensure that the number of global states is finite, the need for which was discussed in Chapter 7). The semantics of time are based on Time Petri Nets and are essentially the same as those of the *PEW*. Only system activity modules are supported, and transitions with

`WHEN` clauses may not have `PRIORITY` clauses.

The simulation capabilities of *ESTIM* are designed for protocol testing rather than performance evaluation. The simulator is thus highly interactive, with the user being able to control the selection of transitions to be executed, display the values of variables, and so on.

Verification is done by building a global reachability graph, and then processing this to extract a simplified automaton representing the service provided by the protocol. Communication through a selected set of channels in the reachability graph is used as the basis for the derived service automaton. This automaton can then be checked by the user to see if it provides the expected service.

10.2 The Estelle Development Toolset *EDT*

The *EDT* system was developed by Bull S.A. in cooperation with and Marben, which has since been extended by the French National Telecommunications Institute (INT). *EDT* consists of several components:

- a *syntax-oriented editor* for Estelle (this was rejected by users for being 'unfriendly');
- an *Estelle compiler* which compiles to an intermediate code;
- a prototype *driver generator* which generates an underlying layer according to attributes such as whether interactions can be lost or rearranged, and an upper user layer which allows the user to interactively construct and send interactions to the protocol;
- a *back-end* and *run-time library* to generate BSD Unix-based C code for implementation or simulation;
- a *simulator/debugger* which includes the ability to backtrack and replay;

The basic architecture of the *EDT* system is shown in Figure 18.

EDT relies on Unix to provide concurrency and inter-process communication, and uses Unix `sockets` to provide the underlying network service when generating implementations.

The simulator/debugger provides a powerful environment for testing the protocol. The biggest drawback is the user-interface, which is teletype-oriented and uses a large set of commands with obscure names. A graphical front-end would simplify the use of the debugger enormously; such a front-end is currently being developed.

10.3 *Veda*

Veda [JMG88] is a simulation tool for Estelle. Specifications are compiled and linked with a simulation kernel to form executable code (actually Pascal, which runs on a Pascal virtual machine).

Figure 18: Architecture of the *EDT* System

In *Vedá*, special *observer* processes can be created, which have global access to the system information. Invariants can be used in observers to assist in validation. These observers must be specified along with the protocol specification by the user of the system.

The *Vedá* compiler was implemented in Prolog, making it slow. However, separate compilation is supported, and *Vedá* can compile large specifications.

Although no recent literature has appeared on *Vedá*, it seems that *Vedá* now supports verification for a subset of Estelle. This was determined from e-mail communication with one of the developers, but no details were available.

Vedá has many of the same goals as the early versions of the *PEW*, when the *PEW* provided no analytical performance methods. It is interesting that the implementation of *Vedá* is thus so different from that of the *PEW*. The use of Prolog has eased the development of a verification engine for *Vedá*.

Part IV

Examples and Discussion

Chapter 11

The Alternating Bit Protocol

In this chapter we apply the protocol performance evaluation techniques described in the thesis to th commonly known *Alternating Bit* protocol, a simple stop-and-wait protocol with a positive acknowledgement. We first use the *PEW* to construct a simple imbedded Markov chain model of this protocol, and use this to obtain performance estimates. Following this we obtain performance predictions using the behaviour graph technique. This requires a minor modification of the specification, from prevent timeouts from occurring before messages have been sent.

The results are compared against those obtained from execution of the same specifications, and an indication of the time taken by each method is given.

The execution times given in this and the next chapter were all obtained using a 25MHz 386-based PC under MS-DOS. The times are given only approximately; this is because other factors, such as efficiency of the code produced by the C compiler, all have an effect on these times, and the values given are intended more to show the differences (sometimes many orders of magnitude) between the simulation and analytical techniques.

11.1 The Protocol

The AB protocol specification is given in the Appendix. The version shown is the version used for the behaviour graph, which includes some code to enforce channel blocking and thus prevent timeouts from occurring before a message has finished being sent (a pathological condition that would cause the channel queues to grow infinitely). These restrictions were not needed in the version used for the semi-Markov model approach.

The AB protocol is a stop-and-wait protocol; that is, after sending a message it will stop and wait for a positive acknowledgement before sending the next message. There can never be more than one outstanding acknowledgement; nontheless, some sequence information is required to distinguish transmission of a new message from a retransmission of the last message. To accomplish this, the

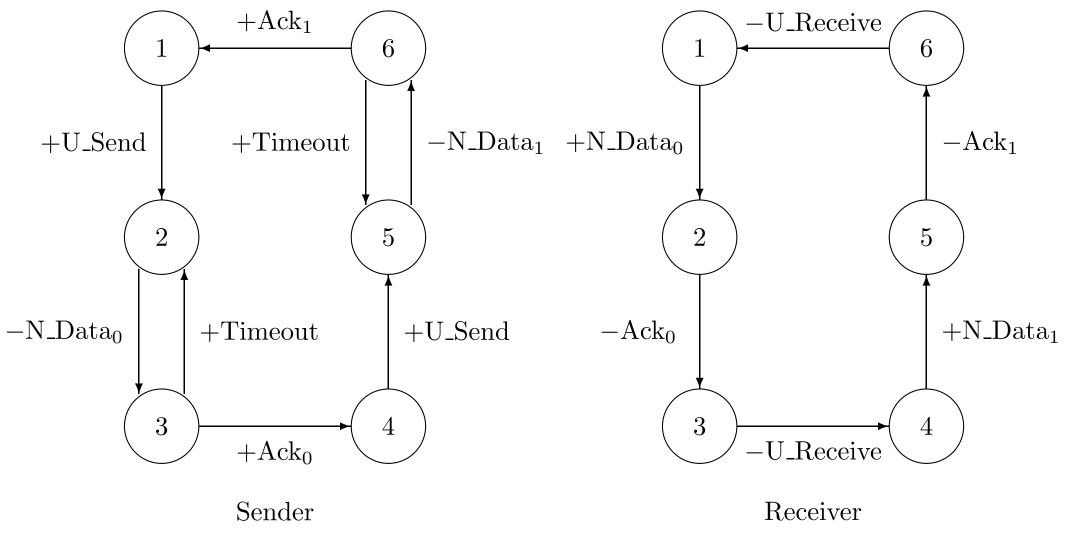

Figure 19: AB-protocol Send and Receive FSMs

sequence numbers 0 and 1 are included within messages and acknowledgements, hence the name *Alternating Bit.* FSM representations of the sender and receiver processes in the AB protocol are shown in Figure 19. Although the sender and receiver are separate, they can be combined into a single process, shown in Figure 20.

The AB protocol specification used for the semi-Markov model is based on this combined FSM, with the addition of a `send indication` being sent from the protocol to the user on receipt of an in-sequence acknowledgement, and duplicate acknowledgements or out-of-sequence data being discarded. The user entity will output another `send request` upon receipt of a `send indication`. This means the protocol is operating at its maximum possible throughput for a particular configuration.

The specification used for the behaviour graph approach consists of a top-level process, and three child processes, one for the protocol, one for the unreliable provider medium, and one for the user application. This specification is for an AB protocol entity in a *loopback* mode; that is, messages sent by the protocol to the provider process are returned to the protocol. The protocol is thus effectively communicating with a mirror image of itself. Using such a loopback specification results in a considerable reduction in the size of the behaviour graph and the time required to construct it.

To enforce channel blocking, a second channel is used to indicate that the provider has completed transmission of a data frame, by `OUTPUT`ing a `sentData` indication. The protocol includes a transition which dequeues these `sentData` indications from the provider and sets a flag. Only once this flag is set does a transition fire which starts the timeout clock running.

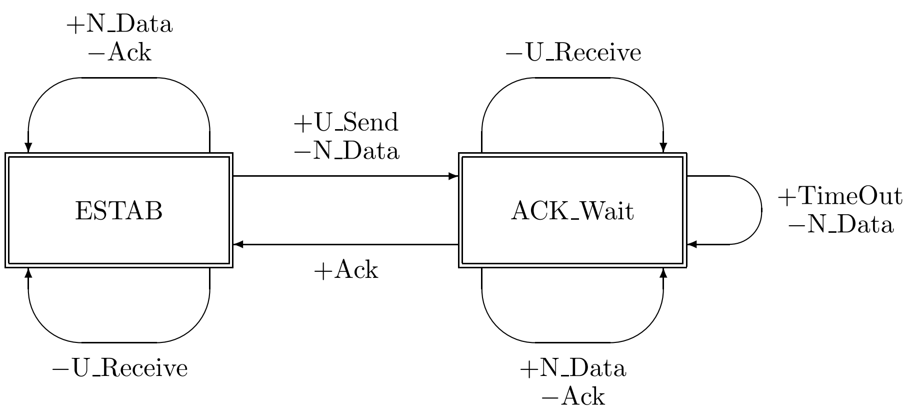

Figure 20: Combined and reduced AB-protocol FSM

11.2 Using a Stochastic Model

To simplify the derivation of mapping functions, we needed to make the relationships between the specification's parameters and its resulting behaviour as simple as possible. A large timeout value was chosen to ensure that timeouts were always due to a message loss, and not due to the timeout value being too short. Deterministic scheduling was also used, and the protocol transitions were split into two groups with the *PEW* compiler's transition grouping directive. The split was made so that we could obtain matrices only for the important transitions in the sender. The receiver transitions have no delays, and their effect on the sender is limited to the delay and loss probabilities of ACK messages, which should in any case be reflected in the sender, as it blocks until the ACK is received. The receiver transitions could thus be ignored, as could the transition to discard duplicate acknowledgements.

Using a timeout value of 12, a 20% probability of loss of data frames and a 5% probability of loss of acknowledgements, we executed the specification for 5000 time units. This took approximately seven minutes. The resulting transition sequence count matrix T, stochastic matrix P, and holding time matrix D delay matrices were:

$$T = \begin{pmatrix} 0 & 154 & 524 \\ 0 & 43 & 154 \\ 678 & 0 & 0 \end{pmatrix} P = \begin{pmatrix} - & 0.23 & 0.77 \\ - & 0.22 & 0.78 \\ 0 & - & - \end{pmatrix} D = \begin{pmatrix} - & 12 & 4.45 \\ - & 12 & 3.35 \\ 0 & - & - \end{pmatrix}$$

From these matrices, we determined that there is a trivial relationship between the timeout value and the second column of the holding time matrix. Furthermore, the stochastic matrix values in the third column are close to 0.76, which is the probability of a message/acknowledgement pair being sent and received without loss (that is, 0.8 * 0.95, where 0.8 is the probability of the message being sent successfully, and 0.95 is the probability of the acknowledgement being sent successfully).

Figure 21: Alternating Bit Protocol Model

This immediately suggested how we could modify these matrices to reflect changes in the timeout and reliability parameters. The model we used is shown in Figure 21. The arc delays and probabilities are shown in Table 1.

The model as shown in the figure has been slightly refined. If we make the timeout value shorter, there is a certain probability that we will time out before an acknowledgement is received, even though neither the message nor the acknowledgement were lost. We need to incorporate the probability of this happening into the model if we are to predict the behaviour of the system as timeouts become shorter.

Arc	Probability	Delay
A1	$1 - P[A4]$	Timeout
A2	$1 - P[A3]$	Timeout
A3	$0.76 * P[t > rtd] * P[t > (2 * rtd)]$	3.35
A4	$0.76 * P[t > rtd]$	4.45
A5	1	0

Table 1: Arc Probabilities and Delays in the Model

Figure 22: Measured and Predicted Values (Exponential Delays)

We thus scale the probability value $p_{1,3}$ by $P[timeout > rtd]$, where rtd is the round-trip delay of a data frame and acknowledgement. Similarly, if the round trip delay is longer than twice the timeout value, we will time out twice. We thus scale $p_{2,3}$ by $P[timeout > (2 * rtd)]$.

In a really poorly configured system, there may be even more timeouts before the acknowledgement arrives. However, this is not the case with the examples we have examined. The model shown in the figure is thus the model we use for all of our predictions.

In order to determine the effect that the probability distributions have on the protocol and model, we executed and modelled specifications using Poisson and exponential delays. We experimented with both the reliability parameter associated with data frames, and with the timeout value. In each case, we measured the number of unique frames sent per time unit (the throughput of the protocol), and the number of timeout and retransmit transitions that occurred per time unit. The results are shown graphically in Figures 22, 23, 24, and 25. Note that the measured values were obtained from executions using non-deterministic scheduling.

The measurements took four hours to obtain, while the predictions were obtained in less than one minute!

To obtain the measurements, we wrote a simple program which output data files containing the number of transitions, and the stochastic and holding time matrices, and then invoked the performance analyser to solve the resulting model. Programs such as this have been very useful in producing parametric performance predictions using the performance analyser, and also serve as useful model references.

Figure 23: Measured and Predicted Values (Poisson Delays)

Figure 24: Measured and Predicted Values (Exponential Delays)

Figure 25: Measured and Predicted Values (Poisson Delays)

The model produces good results for timeouts of twelve or greater. Shorter timeouts lead to a fall in throughput, and eventually to the pathological case described earlier. Here the model is less accurate, although it does indicate the trend. Measured results showed the optimal timeout value to be 10, while the model predicted the slightly lower 8.

The model can easily be refined. In particular, we have not examined the nature of the delays associated with the arcs A3 and A4. These delays (A4 in particular) are related to the end-to-end delays for data and acknowledgement frames, but will also be affected by a timeout that is too short.

11.3 Using a Behaviour Graph

The behaviour graph for this specification is shown in Figure 26. The interpreter converged on this graph using a comparison depth of four in about 20 seconds. Using greater comparison depths resulted in the same graph. The total number of nodes explored was 162, and the maximum depth reached was 13. The final graph has 19 nodes.

Figure 27 shows the same behaviour graph parameterised for a timeout value of 12. The fractional numbers labelling the arcs are the probabilities, while the integer values are the delays. After being parameterised, and removing user and provider transitions and empty nodes, the graph is reduced to 10 nodes.

Each execution of the graph processor for a particular set of parameters took about one second, while the executions of the specification (each for 1000 units of simulated time) took about two

Figure 26: Behaviour Graph of the AB Specification

Figure 27: Parameterised Behaviour Graph of the AB Specification

Figure 28: Selected Measured Transition Throughputs (Poisson Delays)

minutes each. The protocol was executed using both constant and Poisson distributed provider delays, and the results compared to those predicted. The resulting transition throughput rates are shown in Figures 28, 29 and 30.

The reason for using Poisson-distributed provider delays is to show that constant delays can still give useful results. While the protocol throughput is slightly higher when constant delays are used, the differences are not great, and the trends are identical.

As can be seen from these figures, the behaviour graph technique gives excellent performance predictions, and can do so faster than by running simulations, particularly once the graph has been built, as this is the most time- and space-consuming part of the method but needs to be done once only.

11.4 Detecting Errors in a Specification

The behaviour graph method can also be used to detect problems in a protocol specification. To demonstrate this, the second specification was altered to introduce livelock, by modifying the last transition, which resends the last acknowledgment upon receipt of an out of sequence frame. The resend was removed, so the transition simply discards the out of sequence frame.

The resulting behaviour graph is shown in Figure 31. The livelock can be clearly seen in the bottom five nodes of the graph. The only transition in the protocol which can execute once we enter this portion of the graph are `a1` (`sentData` indication received), `a3` (timeout and resend), and `a7`

Figure 29: Selected Measured Transition Throughputs (Constant Delays)

Figure 30: Selected Predicted Transition Throughputs (Constant Delays)

(dequeue out-of-sequence data); the throughput thus drops to zero (which is an alternative way of detecting the livelock).

It is not necessary to draw the behaviour graph to see this type of situation. The *PEW* automatically performs reachability analysis of a behaviour graph when it is complete, and warns the user of possible livelock and deadlock situations if such subgraphs are found which exclude one or more of the transitions in the system.

Figure 31: Behaviour Graph of AB Specification with Livelock

Chapter 12

The SLAP Protocol

12.1 The Protocol

In this chapter we consider a more sophisticated protocol, the *SLAP* (Simple Link Access Procedure) protocol. This protocol is based on the X.25 link layer (LAP/B) protocol, with some simplifications. This is a sliding window protocol which allows for frame loss and frame corruption. The protocol provides a reliable end-to-end service to the user process.

Differences from the standard LAP/B protocol include:

- No connection management is included, as we are interested purely in the throughput once a connection has been established;
- The frame reject condition has been simplified; when an out-of-sequence frame is received, any other data frames that are awaiting processing are discarded, a REJ frame is sent, and we immediately return to the normal condition;
- No busy condition is included. This is because the busy condition depends on higher layers being unavailable to process frames; we are not interested in these higher layers. Flow control has instead been implemented using a method similar to typical implementations of the network layer of X.25, namely by maintaining a number of credits in the user layer. Each time a frame is successfully sent, the user process is given a new credit. Initially the user has as many credits as there are buffers in the sliding window. When a frame is sent, a credit is used up;
- No P/F bit is used. I frames always get a response, while RR and REJ frames will only get a response if there are I frames to be sent or resent.

Corruption of messages is indicated by the use of a Boolean field `ok` within I frames. This field gets set or cleared randomly by the provider process. Corrupt I frames are treated as out-of-sequence frames.

The timeout mechanism for this protocol has some similarity with that used in the loopback AB protocol. That is, the timeout clock will only start running once the I frame which causes it to start has been sent. This allows us to use the behaviour graph technique with no modification of the protocol (other than once again making use of a loopback specification). We apply the BG method first in this chapter.

The semi-Markov model approach is more difficult with this protocol. This is because the degradation of the protocol's throughput is almost linear with increasing timeout. For timeout values of less than ten, the protocol is very unstable, with great variations in throughput possible between each side or between different executions. The stochastic matrix also varies considerably in this range, with transition sequences that normally never occur occurring sporadically or even in large proportions. Because of this, we have not tried to model the behaviour of the protocol in this range.

The protocol specification is given in Appendix C. This is the full version with two user and two protocol processes. The specification used for the behaviour graph analysis was identical except that only one user and one protocol process were used, and the provider provided a loopback service.

12.2 Using a Behaviour Graph

The *PEW* built a behaviour graph for the SLAP protocol in under five minutes[1], using a node comparison depth of four. Comparison depths of five and six were also used; these resulted in the same graph. There is a discrepancy between the predicted and measured results for a timeout value of four that may be explained by the graph not being entirely correct; it could be that a greater node comparison depth would reveal some slight change. However, the extra time that would be required to perform this check is not merited, as a timeout value of four in any case lies in the unstable region of the protocol, and is definitely too short.

In building the behaviour graph, the *PEW* explored 872 nodes, with the greatest depth reached being fifteen. The final behaviour graph had 52 nodes before parameterisation.

There are six parameters that can be experimented with with this graph, namely the three loss probabilities associated with the different frame types, the timeout delay, and the provider delays associated with the sending of I frames and control frames. As an example, we modified the timeout delay from 2 through to 40, in steps of two. We also measured the performance of the specification using simulations of 10000 units each, using both Poisson and constant delays. The simulation results for Poisson delays are shown in Figure 32, those for constant delays in Figure 33, and the predicted results are shown in Figure 34. Each measurement took about 7 minutes, while each prediction took about 50 seconds. Note that message corruption *was* allowed in the simulations, but was *not* allowed in the behaviour graph analysis and predictions, as the *PEW* does not deal with nondeterminism caused by the use of the various random-number generation functions that have been added to the

[1]This is quite impressive when one considers the inefficient way in which the *PEW* currently builds such graphs.

Figure 32: Measured Transition Throughputs for Loopback SLAP Protocol using Poisson Provider Delays

Figure 33: Measured Transition Throughputs for Loopback SLAP Protocol using Constant Provider Delays

Figure 34: Predicted Transition Throughputs for Loopback SLAP Protocol using Constant Provider Delays

PEW's Estelle implementation. We would thus expect the measured throughputs to be slightly lower than the predicted ones, although this difference is very small and is not noticeable from the graphs.

12.3 Using a Stochastic Model

To build a stochastic model of this protocol is far less straightforward. We attempted to follow the same strategy as with the AB protocol, and concentrate on transitions related to sending. The transitions chosen were:

- The provider becoming ready. This was chosen as this transition is responsible for starting the timeout clock running;
- The timeout transition itself;
- A send request being received from the user. As an acknowledgment can be handled in several transitions, but always results in the user issuing another send request, this transition was chosen instead of the others to indicate that a frame was successfully transmitted;
- The two transitions which `OUTPUT` I frames. The first of these is an unsolicited transfer (low priority), while the second is in response to peer events, and is used as an acknowledgement

as well as an I frame transmission. This has a higher priority; thus, unsolicited transmission of I frames will only occur if there are no frames from the peer that must be dealt with.

These are the first five body transitions of the protocol specification as given in the Appendix.

Executing the specification with a timeout value of twenty resulted in the stochastic and holding time matrices:

$$P = \begin{pmatrix} 0.171 & 0.073 & 0.273 & 0.319 & 0.163 \\ 0.186 & 0.000 & 0.000 & 0.814 & 0.000 \\ 0.852 & 0.000 & 0.001 & 0.147 & 0.000 \\ 0.681 & 0.000 & 0.320 & 0.000 & 0.000 \\ 0.427 & 0.000 & 0.573 & 0.000 & 0.000 \end{pmatrix} D = \begin{pmatrix} 4.0 & 12. & 1. & 0.025 & 0. \\ 0.5 & - & - & 0 & - \\ 2.8 & - & 0 & 0 & - \\ 4.0 & - & 0.25 & - & - \\ 3.5 & - & 1.3 & - & - \end{pmatrix}$$

Unfortunately, there is no clear correspondence here between specification parameters and the matrices. We thus executed the same specification with a timeout of 40, to see how the matrix would change. The new stochastic matrix is:

$$P = \begin{pmatrix} 0.220 & 0.073 & 0.236 & 0.333 & 0.140 \\ 0.115 & 0.000 & 0.000 & 0.884 & 0.000 \\ 0.862 & 0.000 & 0.002 & 0.136 & 0.000 \\ 0.621 & 0.000 & 0.380 & 0.000 & 0.000 \\ 0.433 & 0.000 & 0.567 & 0.000 & 0.000 \end{pmatrix}$$

The only significant changes in the delay matrix were in the first two entries in the first row. The delay of 4 changed to 7, while that of 12 changed to 23. In each case this represents a doubling of the delay (ignoring the slight error thus introduced). The major differences in the stochastic matrix were in the first, second, and fourth rows.

Unlike the AB model where the relationships between loss probabilities and probabilities in the stochastic matrix were obvious, in this case they are not. For our model, we decided to keep the third and fifth rows of the matrix constant, while making the entries in the other rows linear functions of the timeout value. We also made the two delays that change linearly dependent on the timeout, while keeping the rest constant. The stochastic matrix we derived (where t is the timeout), is:

$$P = \begin{pmatrix} .1 + .0035t & .0735 & .32 - .0022t & .32 & .19 - .0013t \\ 1 - p_{2,4} & 0 & 0 & 0.74 + .00365t & 0 \\ 0.85 & 0 & 0 & 0.15 & 0 \\ 1 - p_{4,3} & 0 & 0.25 + .0325t & 0 & 0 \\ 0.426 & 0 & 0.574 & 0 & 0 \end{pmatrix}$$

The holding time matrix is:

Figure 35: Measured Throughputs of Selected Transitions in SLAP Protocol

$$P = \begin{pmatrix} .2t & .6t & 1 & .0025 & 0 \\ 0.5 & - & - & 0 & - \\ 2.8 & - & 0 & 0 & - \\ 4 & - & 0.25 & - & - \\ 3.5 & - & 1.3 & - & - \end{pmatrix}$$

The measured and predicted results for the specification are shown in Figures 35 and 36. Each measurement took 15 minutes, while each prediction took less than one second. The predicted results are good, particularly for timeout values of greater than 10 (below this, the protocol's performance degrades, something the model fails to predict). Nontheless, the example is disappointing due to the near-linear behaviour for larger timeouts.

12.4 Conclusion

In the case of this protocol, the behaviour graph technique offers a better set of results that the stochastic model. The fact that the behaviour of the protocol is nearly linear leads to the question of whether the stochastic model produced reasonable results simply because of the linear interpolations used on delays and probabilities in the matrix. In defense, it should be noted that the throughputs tend asymptotically to zero rather than degrading linearly, and this can be seen in the predictions as well.

Figure 36: Predicted Throughputs of Selected Transitions in SLAP Protocol

One point that should be noted is that the model as given cannot be used for any arbitrary timeout value. Very large timeout values will result in some of the probabilities becoming negative, so there is a limit to its usefulness. Nontheless, the example serves to illustrate how we can build stochastic models to approximate the performance of a more complex protocol and obtain performance predictions much faster than running simulations.

Chapter 13

Discussion and Conclusion

This thesis has presented a tool for protocol engineering based on Estelle specifications. This tool is but one of many, some of which were described in chapter 10. There are a number of features of the *PEW* which, together or individually, distinguish it from other tools:

- the use of Estelle as the description technique;
- the use of a meta-implementation of the specification language;
- the extra facilities included to aid in modelling characteristics of protocol systems;
- the simulation capabilities;
- the generation of information for use in building stochastic models of specified systems;
- the stochastic modelling method itself;
- the behaviour graph technique for performance prediction and protocol checking.

In the next section, we summarise the advantages and disadvantages of the performance evaluation methods we have presented. As some of the weaknesses are due to the use of Estelle, a brief critique of Estelle is also given. We end by describing some of the possible future directions in which this work (and the *PEW*) can be taken.

13.1 Evaluation of the Methods

13.1.1 The Semi-Markov Model Method

The approach of deriving semi-Markov models from the transition sequence matrices output by the *PEW* has the following advantages:

- the models are generally small (particularly as only a subset of transitions is usually chosen), and predictions can thus be made very rapidly in comparison with simulation;
- the technique is general and places no absolute restrictions on the systems that can be handled (on the other hand, imposing restrictions such as deterministic scheduling on the systems may simplify the task of determining mapping functions);
- the entire approach can, in principle, be automated, by using interpolation between the known individual holding time and stochastic matrix entries of a small set of matrices measured from executions. The technique as currently implemented in the *PEW* is a semi-automatic one, which does have the benefit of allowing a more heuristic approach to determine the mapping functions than just using interpolation.

The drawback of the approach is that suitable mapping functions must be determined and there is no guaranteed algorithm for this; interpolation can be used as an approximate algorithm, while our approach has generally been to perform a small set of executions and then examine the results and construct approximate mapping funxtions by hand. This is an area in which further research may yield interesting and useful results.

13.1.2 The Behaviour Graph Method

The behaviour graph method has the following advantages:

- it allows the checking of safety properties as a side-effect;
- it produces highly accurate performance predictions;
- it works with implementation-oriented specifications.

The drawbacks of the method are:

- The state-explosion problem. Furthermore, the implementation of BG building in the *PEW* is highly inefficient, which forced us to use loopback mode specifications in order to obtain a graph in a reasonable amount of time.
- The modifications that must be made to a specification. These are mostly required to ensure the number of global system states is finite. If a more suitable language than Estelle were used many of these restrictions could possibly be lifted.
- The constant delay restriction when parameterising the graph. Again, this is to reduce the state explosion problem. The method could easily be extended to geometrically distributed delays but the resulting parameterised graphs would be much larger.

13.2 Protocol Engineering and Estelle

As there are now international standards for FDTs, it is more appropriate to use one of these standards than to develop a proprietary system[1] There are an increasing number of protocols being specified in Estelle by researchers around the world; these existing specifications can be used by the *PEW* with a minimal amount of modification.

However, there are a number of characteristics of Estelle that are either restrictive or 'unnatural' when it comes to specifying, and more particularly, modelling communication protocols. The *PEW* has attempted to address a number of these by using compiler directives, and more can still be done.

In our experience with Estelle, we have found the following restrictions the most onerous:

- The FIFO-only nature of interaction queues;
- The asynchronous message passing mechanism;
- The fact that the queues are in principle unbounded;
- The use of Pascal as a language base;[2]
- The fact that Estelle statements are not allowed within procedures and functions (that is, procedures and functions in a specification may use Pascal constructs only).

Some of these deserve further elaboration. The FIFO nature of the queues has been dealt with in the *PEW* by introducing priority and random queueing as options.

The problems that can result from the unbounded nature of queues and the asynchronous message passing mechanism have been described before. The *PEW* includes a command line argument which will enforce a degree of synchronism on specifications, by restricting the capacity of queues to single interactions. This is unwieldy to implement[3], and is not an ideal solution to the problem.

The use of Pascal as a language base makes it difficult to specify features such as variable length data frames in such a way that the specification can still be compiled and executed or translated into an implementation. Variable length frames are specified in Estelle using the `ANY` constant type. This is adequate for specificying the variable length frame, but unsuitable for handling by an Estelle compiler.

The restrictions imposed upon procedures and functions are more an irritation than serious drawbacks. There have been several occasions where we could have considerably simplified Estelle specifications had we been able to include `OUTPUT` statements within procedures.

[1]Actually, this is an issue on which not everyone agrees, but there are many advantages to standardising on open standards.

[2]Again, we must tread lightly here!

[3]We have to inspect not only transition clauses, but also whether they have `OUTPUT` statements which attempt to output to non-empty queues, before we can determine which transitions are fireable.

In Estelle's defense, it must be noted that it was designed as a specification language, and not for implementation or simulation. The fact that it has been used for these other applications is a favourable comment on the language.

Of course, the fact that Estelle is an ISO standard does not make it a static entity. The ISO committees regularly consider submissions for modifications to standards, and there have been a number of suggestions put forward for Estelle, including the rendezvous mechanism of Estelle*. Some of these criticisms may no longer apply to future versions of Estelle.

13.3 Further Work

Work on the *PEW* is likely to continue. An X/Windows-based specification system that has been partially implemented should be completed, making the *PEW* a more high-level CASE tool. It is likely that work will be done on a verification system as well; the behaviour graph technique lends itself towards such extensions. Furthermore, there are sufficient similarities between Estelle and SDL to make it possible to use the same underlying VM engine for executing SDL specifications as well; this is another possible future extension.

The work that has been described in this thesis should thus be seen in the context of the long-term goals of the *PEW* project. We would like to build a protocol engineering tool capable of protocol specification, testing, verification and performance analysis, all using a single formal description technique. The foundation for such a tool has now been laid. The *PEW* provides a compiler and interpreter which allows the syntactic and semantic checking of a specification, and its performance evaluation through various approaches.

The behaviour graph technique shows that the fields of verification (using state exploration techniques) and performance analysis may be fruitfully combined. The high cost of verification means that other performance evaluation strategies will still be important, but if performance figures can be obtained from a global state graph of a system for no extra cost during the state-exploration process, this is useful indeed.

Other directions in which the *PEW* can be taken include:

- the automatic generation of user and provider processes, given the characteristics that these processes should have.
- when deadlock or livelock is detected in a behaviour graph, the sequence of events that leads to this situation could be printed out;
- the imbedded Markov chain models currently supported by the *PEW* represent just one possible approach. Other stochastic modelling methods could possibly be used in conjunction with the *PEW* as well.

Part V

Appendices

Appendix A

The Alternating Bit Protocol

```
{ THE ALTERNATING BIT PROTOCOL: TWO MACHINE IMPLEMENTATION }

SPECIFICATION AB_Example; TIMESCALE MILLISECONDS;

CONST
        low = 1;
        high = 2;
        Timeout = 12;
        data_packet_len = 1;    { for simplicity                }

TYPE
        cep_type = low..high;
        udata_type = array[1..data_packet_len] of char;
        seq_type = 0..1;
        pdata_type = RECORD
            conn: cep_type;
            data: udata_type;
            seq: seq_type;
        END;

{**********************************************************************

        Channel definitions: there are two types of channels, one to
        connect user processes to AB processes, and another to connect
        AB processes into the provider process.

**********************************************************************}

CHANNEL u_access_point(user, protocol);
        BY user:
            send_request(udata:udata_type);
        BY protocol:
            receive_response(udata:udata_type);
        send_indication;

CHANNEL p_access_point(protocol, provider);
        BY protocol:
            data_request(pdata:pdata_type);
            ACK_request(pdata:pdata_type);
        BY provider:
            data_indication(pdata:pdata_type);
            ACK_indication(pdata:pdata_type);


{**********************************************************************

        PROVIDER MODULE
        The provider module passes along requests from one AB process to
        the other, not necessarily reliably.

**********************************************************************}

MODULE provider_type SYSTEMPROCESS;
        IP p: ARRAY[cep_type] OF p_access_point(provider) INDIVIDUAL QUEUE;
END;

BODY provider_body FOR provider_type;
TYPE
        provider_state = (Available, SendingData, SendingACK);
VAR
        Route12state, Route21state: provider_state;
        Route12Delay, Route21Delay:INTEGER;

INITIALIZE
        NAME p0:
        BEGIN
            Route12state := Available;
            Route21state := Available;
        END;

TRANS {************* Transitions for 1->2 ******************}
        PROVIDED Route12state = Available
            WHEN p[1].ACK_request
                NAME p1:
                BEGIN
                    { ACK transmissions are 95% reliable }
                    {$R95} OUTPUT p[2].ACK_indication(pdata);
                    Route12state := SendingACK;
                    Route12Delay := PRANDOM(1,1,7);
                END;
            WHEN p[1].data_request
                NAME p2:
                BEGIN
                    { Data transmissions are 90% reliable }
                    {$R90} OUTPUT p[2].data_indication(pdata);
                    Route12state := SendingData;
                    Route12Delay := PRANDOM(1,3,7);
                END;
        PROVIDED Route12state = SendingACK
            DELAY (Route12Delay)
                NAME p3:
                BEGIN
                    Route12state := Available;
                END;
        PROVIDED Route12state = SendingData
            DELAY (Route12Delay)
                NAME p4:
                BEGIN
                    Route12state := Available;
                END;

TRANS {************* Transitions for 2->1 ******************}
        PROVIDED Route21state = Available
            WHEN p[2].ACK_request
                NAME p5:
                BEGIN
                    { ACK transmissions are 95% reliable }
                    {$R95} OUTPUT p[1].ACK_indication(pdata);
                    Route21state := SendingACK;
                    Route21Delay := PRANDOM(1,1,7);
                END;
            WHEN p[2].data_request
                NAME p6:
                BEGIN
                    { Data transmissions are 90% reliable }
                    {$R90} OUTPUT p[1].data_indication(pdata);
                    Route21state := SendingData;
                    Route21Delay := PRANDOM(1,3,7);
                END;
        PROVIDED Route21state = SendingACK
            DELAY (Route21Delay)
                NAME p7:
                BEGIN
                    Route21state := Available;
                END;
        PROVIDED Route21state = SendingData
            DELAY (Route21Delay)
                NAME p8:
                BEGIN
                    Route21state := Available;
                END;
END;


{**********************************************************************

        ALTERNATING BIT MODULE BODY

**********************************************************************}

MODULE alternating_bit_type SYSTEMPROCESS(conn_end_pt_id : cep_type);
        IP  u: u_access_point(protocol) COMMON QUEUE;
            p: p_access_point(protocol) INDIVIDUAL QUEUE;
END;

BODY alternating_bit_body FOR alternating_bit_type;
CONST
    buf_size = 3;
TYPE
        msg_type = RECORD
            msgdata: udata_type;
            msgseq: seq_type;
        END;
VAR
    send_seq, recv_seq: seq_type;
    send_msg, recv_msg: msg_type;
    recv_buff: ARRAY[1..buf_size] OF pdata_type;
    ppdu, ppdu_ack: pdata_type;
    buf_in, buf_out, buf_cnt:INTEGER;
STATE
        ACK_Wait, Estab;

STATESET
        Either = [ACK_Wait, Estab];

PROCEDURE format_data_packet(msg: msg_type; VAR ppdu:pdata_type);
BEGIN
        ppdu.conn := conn_end_pt_id;
        ppdu.data := msg.msgdata;
        ppdu.seq := msg.msgseq;
END;

PROCEDURE format_ack_packet(msg: pdata_type; VAR ppdu:pdata_type);
BEGIN
        ppdu.conn := conn_end_pt_id;
        ppdu.seq := msg.seq;
END;

INITIALIZE
        TO Estab
            NAME a0:
            BEGIN
                send_seq        := 0;
                recv_seq        := 0;
                buf_cnt     := 0;
                buf_in      := 1;
                buf_out     := 1;
            END;

{****************** SEND TRANSITIONS ***************************}

{ Get user data, buffer and send }

TRANS FROM Estab TO ACK_Wait
        WHEN u.send_request
        NAME a1:
            BEGIN
                send_msg.msgdata:= udata;
                send_msg.msgseq := send_seq;
                format_data_packet(send_msg,ppdu);
                OUTPUT p.data_request(ppdu);
            END;

{  Timeout and retransmit...}

FROM ACK_Wait TO same
        DELAY (Timeout)
        NAME a2:
            BEGIN
                format_data_packet(send_msg,ppdu);
                OUTPUT p.data_request(ppdu);
            END;

{ Receive ACK for packet; unbuffer and return to ESTAB state }

TRANS
WHEN p.ACK_indication
        PROVIDED (pdata.seq = send_seq)
            FROM ACK_Wait TO ESTAB
            NAME a3:
                BEGIN
                    send_seq := (send_seq+1) MOD 2;
                    OUTPUT u.send_indication;
                END;

{$T Duplicate ACK }

WHEN p.ACK_indication
        PROVIDED (pdata.seq <> send_seq)
        NAME a4:
            BEGIN
            END;


{ Received correct data; buffer it for user and send ACK }

WHEN p.data_indication
        PROVIDED (pdata.seq = recv_seq) AND (buf_cnt<buf_size)
        NAME a5:
            BEGIN
            buf_cnt := buf_cnt + 1;
            recv_buff[buf_in] := pdata;
            buf_in := buf_in+1;
            IF (buf_in>buf_size) THEN buf_in := 1;
                format_ack_packet(pdata,ppdu_ack);
                OUTPUT p.ACK_request(ppdu_ack); { send ACK }
                recv_seq := (recv_seq+1) MOD 2;
            END;
TRANS
    PROVIDED buf_cnt>0
        NAME a6:
        BEGIN
            OUTPUT u.receive_response(recv_buff[buf_out].data);
            buf_cnt := buf_cnt - 1;
            buf_out := buf_out+1;
            IF (buf_out>buf_size) THEN buf_out := 1;
        END;

{ Out of sequence data }

TRANS
WHEN p.data_indication
        PROVIDED (pdata.seq<>recv_seq)
        NAME a7:
            BEGIN
                OUTPUT p.ACK_request(ppdu_ack); { Resend ACK }
            END;

END; { OF AB body }

{*********************************************************************
        USER MODULE BODY

*********************************************************************}

MODULE user_type SYSTEMPROCESS(conn_end_pt_id: cep_type);
        IP u: u_access_point(user) INDIVIDUAL QUEUE;
END;

BODY user_body FOR user_type;
VAR
        data_pack: udata_type;
    first: BOOLEAN;
INITIALIZE
    BEGIN
        first := TRUE;
    END;
TRANS
    PROVIDED first
        Name u0:
        BEGIN
            first := FALSE;
            OUTPUT u.send_request(data_pack);
        END;
TRANS
        WHEN u.receive_response  { data received from AB }
        NAME u1:
            BEGIN
                { just dequeue it }
            END;

    WHEN u.send_indication { Just send another }
        NAME u2:
        BEGIN
                OUTPUT u.send_request(data_pack);
        END;
END;

{*********************************************************************

        SPECIFICATION BODY

*********************************************************************}

MODVAR
        user: ARRAY[cep_type] OF user_type;
        alternating_bit: ARRAY[cep_type] OF alternating_bit_type;
        provider: provider_type;

INITIALIZE
        VAR
            child: cep_type;
        BEGIN
            INIT provider WITH provider_body;
            FOR child:=low TO high DO BEGIN
                INIT user[child] WITH user_body(child);
                INIT alternating_bit[child] WITH alternating_bit_body(child);
                CONNECT user[child].u TO alternating_bit[child].u;
                CONNECT alternating_bit[child].p TO provider.p[child];
            END;
        END;
END.
```

Appendix B

buildmat.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define LIMIT 1000l

#ifdef UNIX
extern double drand48();
#else
#define drand48()   ( ((double)rand()) / ((double)RAND_MAX) )
#endif

double myrandom(range)
short range;
{
    return (double)(drand48()*(double)range);
}


short Dis_Exponential(Min, Max, mean)
short   Min,
    Max,
    mean;
{
    short rtn;
    double Mean = (double)mean;
    do  {
        double tmp = (1.+myrandom(1000))/1000.;
        rtn = (short)(-Mean*log(tmp));
    } while (rtn<Min || rtn>Max);
    return rtn;
}

short Dis_Poisson(Min, Max, mean)
short   Min,
    Max,
    mean;
{
    double b=exp((double)-mean), tr;
    short rtn;
    do  {
        rtn=-1;
        tr=1.;
        do  {
            tr *= drand48();
            rtn++;
        } while (tr>b);
    } while (rtn<Min || rtn>Max);
    return rtn;
}
```

```
short Dis_Geometric(Min, Max, mean)
short   Min,
    Max,
    mean;
{
    short rtn;
    float expekt = log(1.001-1./((float)mean));
    do  {
        rtn = (short)(.5+(log(.001+drand48())/expekt));
    } while (rtn<Min || rtn>Max);
    return rtn;
}

float bucket[120];
float Bucket[120];

buildBucket(int typ, int min1, int mean1, int max1,
                     int min2, int mean2, int max2) {
    int i, t;
    float sum;
    for (i=0;i<120;i++) Bucket[i] = bucket[i]=0.;
    switch(typ) {
        case 1: for (i=0;i<1000;i++)
                bucket[Dis_Poisson(min1,max1,mean1)+
                          Dis_Poisson(min2,max2,mean2)]++;
            break;
        case 2: for (i=0;i<1000;i++)
                bucket[Dis_Exponential(min1,max1,mean1)+
                          Dis_Exponential(min2,max2,mean2)]++;
            break;
        case 3: for (i=0;i<1000;i++)
                bucket[Dis_Geometric(min1,max1,mean1)+
                          Dis_Geometric(min2,max2,mean2)]++;
            break;
    }
    for (i=0;i<120;i++) bucket[i]/=1000.;
    sum = 0;
    for (i=0;i<120;i++) {
        sum += bucket[i];
        Bucket[i] = sum;
    }
}

main() {
    char cmd[20];
    FILE *fp;
    int timeout=12, reldata, i;
    float relACK=.95;
    buildBucket(1,1,1,7,1,3,7);
    fp = fopen("relp.pre","wt");
    for (i=0;i<120;i++)
        if (bucket[i]) fprintf(fp,"%d %.3f %.3f\n",i,bucket[i],Bucket[i]);
    fclose(fp);
    for (timeout=12,reldata=80;reldata<=100;reldata+=2) {
        FILE *fp = fopen("mat.txt","w");
        float p, q;
        p = q = relACK*((float)reldata)/100.;
        p *= Bucket[timeout];
        q *= Bucket[timeout]*Bucket[timeout*2];
        fprintf(fp,"3\n0.00 %f %f  0.00 %d. 4.45\n",1.-p,p,timeout);
        fprintf(fp,"0.00 %f %f  0.00 %d. 3.35\n",1.-q,q,timeout);
        fprintf(fp,"1. 0. 0. 0. 0. 0.\n");
        fprintf(fp,"\nRELIABILITY %d\n\n",reldata);
        fclose(fp);
        sprintf(cmd,"pa -A relp.pre");
        system(cmd);
    }
    for (reldata=90,timeout=6;timeout<=24;timeout+=2) {
        FILE *fp = fopen("mat.txt","w");
        float p, q;
        p = q = relACK*((float)reldata)/100.;
        p *= Bucket[timeout];
        q *= Bucket[timeout]*Bucket[timeout*2];
        fprintf(fp,"3\n0.00 %f %f  0.00 %d. 4.45\n",1.-p,p,timeout);
        fprintf(fp,"0.00 %f %f  0.00 %d. 3.35\n",1.-q,q,timeout);
        fprintf(fp,"1. 0. 0. 0. 0. 0.\n");
        fprintf(fp,"\nTIMEOUT %d\n\n",timeout);
        fclose(fp);
        sprintf(cmd,"pa -A timp.pre");
        system(cmd);
    }
    buildBucket(2,1,1,7,1,3,7);
    fp = fopen("rele.pre","wt");
    for (i=0;i<120;i++)
        if (bucket[i]) fprintf(fp,"%d %.3f %.3f\n",i,bucket[i],Bucket[i]);
    fclose(fp);
    for (timeout=12,reldata=80;reldata<=100;reldata+=2) {
        FILE *fp = fopen("mat.txt","w");
        float p, q;
        p = q = relACK*((float)reldata)/100.;
        p *= Bucket[timeout];
        q *= Bucket[timeout]*Bucket[timeout*2];
        fprintf(fp,"3\n0.00 %f %f  0.00 %d. 4.275\n",1.-p,p,timeout);
        fprintf(fp,"0.00 %f %f  0.00 %d. 2.7.\n",1.-q,q,timeout);
        fprintf(fp,"1. 0. 0. 0. 0. 0.\n");
        fprintf(fp,"\nRELIABILITY %d\n\n",reldata);
        fclose(fp);
        sprintf(cmd,"pa -A rele.pre");
        system(cmd);
    }
    for (reldata=90,timeout=6;timeout<=24;timeout+=2) {
        FILE *fp = fopen("mat.txt","w");
        float p, q;
        p = q = relACK*((float)reldata)/100.;
        p *= Bucket[timeout];
        q *= Bucket[timeout]*Bucket[timeout*2];
        fprintf(fp,"3\n0.00 %f %f  0.00 %d. 4.275\n",1.-p,p,timeout);
        fprintf(fp,"0.00 %f %f  0.00 %d. 2.7\n",1.-q,q,timeout);
        fprintf(fp,"1. 0. 0. 0. 0. 0.\n");
        fprintf(fp,"\nTIMEOUT %d\n\n",timeout);
        fclose(fp);
        sprintf(cmd,"pa -A time.pre");
        system(cmd);
    }

    buildBucket(3,1,1,7,1,3,7);
    fp = fopen("relg.pre","wt");
    for (i=0;i<120;i++)
        if (bucket[i]) fprintf(fp,"%d %.3f %.3f\n",i,bucket[i],Bucket[i]);
    fclose(fp);
    for (timeout=12,reldata=80;reldata<=100;reldata+=2) {
        FILE *fp = fopen("mat.txt","w");
        float p, q;
        p = q = relACK*((float)reldata)/100.;
        p *= Bucket[timeout];
        q *= Bucket[timeout]*Bucket[timeout*2];
        fprintf(fp,"3\n0.00 %f %f  0.00 %d. 3.8\n",1.-p,p,timeout);
        fprintf(fp,"0.00 %f %f  0.00 %d. 1.6\n",1.-q,q,timeout);
        fprintf(fp,"1. 0. 0. 0. 0. 0.\n");
        fprintf(fp,"\nRELIABILITY %d\n\n",reldata);
        fclose(fp);
        sprintf(cmd,"pa -A relg.pre");
        system(cmd);
    }
    for (reldata=80,timeout=6;timeout<=24;timeout+=2) {
        FILE *fp = fopen("mat.txt","w");
        float p, q;
        p = q = relACK*((float)reldata)/100.;
        p *= Bucket[timeout];
        q *= Bucket[timeout]*Bucket[timeout*2];
        fprintf(fp,"3\n0.00 %f %f  0.00 %d. 4.15\n",1.-p,p,timeout);
        fprintf(fp,"0.00 %f %f  0.00 %d. 2.9\n",1.-q,q,timeout);
        fprintf(fp,"1. 0. 0. 0. 0. 0.\n");
        fprintf(fp,"\nTIMEOUT %d\n\n",timeout);
        fclose(fp);
        sprintf(cmd,"pa -A timg.pre");
        system(cmd);
    }
}
```

Appendix C

The SLAP Protocol

```
{*******************************************************

  SLAP - Simple Link Access Procedure/Balanced Protocol

This protocol specification is for a simple link layer
protocol based on LAP-B. The protocol has a similar logical
behaviour to LAP-B, but with the following differences:

i) No unusual error conditions are allowed.

Erroneous frames would be treated as lost. There is thus no
FRMERR frame type, and no error condition.

ii) No BUSY condition

No BUSY condition is supported either, and the user
process is always able to receive frames. Flow control
with the user process is done by giving the user a number
of ‘credits’. The number of outstanding acknowledgements
can be no more than the number of credits. This is the
same as the window size. This type of flow control is more
like the network layer of X.25.

ii) No connection management.

SLAP assumes that a permanent virtual circuit is maintained. This
is more because we are interested in throughput than for any
other reason.

iii) No P/F bit

SLAP does not make use of a poll/final bit. I frames always get
a response, while RR and REJ frames will only get a response
if there are frames to be sent or resent.

This version also allows for message corruption as well as message
loss.

*****************************************************************}

SPECIFICATION SLAP_spec; TIMESCALE SECONDS;

CONST
        ep_low = 1;
        ep_high = 2;
        win_size = 2;
        Buffer_Limit = 1;       { Window_Size-1 }
        Last_Seq_Num = 7;       { 0..7 }
        Timeout = 20;

TYPE
```

```
        cep_type = ep_low..ep_high;
        data_type = integer;
        seq_type = 0..Last_Seq_Num;

{*******************************************************************

        Channel definitions: there are three types of channels, one to
        connect application processes to SLAP processes, another to
        connect SLAP processes into the provider process, and a third
        to indicate to the protocol that the provider is available.

*******************************************************************}

CHANNEL user_slap_chan(user_role, slap_role);
        BY user_role:
            send_request(data:data_type);
        BY slap_role:
            send_indication;
            receive_response;

CHANNEL slap_net_chan(slap_role, net_role);
        BY slap_role, net_role:
            RR(send_seq:seq_type; recv_seq:seq_type);
            REJ(send_seq:seq_type; recv_seq:seq_type);
            I(send_seq:seq_type; recv_seq:seq_type; data:data_type; ok:BOOLEAN);

CHANNEL ProvState_chan(slap_role, net_role);
        BY net_role:
            provReady;

{******************************************************************

        PROVIDER MODULE

        The provider module passes along requests from one protocol process
        to the other, not necessarily reliably.

******************************************************************}

MODULE provider_type SYSTEMPROCESS;
        IP  n: ARRAY[cep_type] OF slap_net_chan(net_role) INDIVIDUAL QUEUE;
            pp: ARRAY[cep_type] OF provState_chan(net_role) INDIVIDUAL QUEUE;
END;

BODY provider_body FOR provider_type;
TYPE
        provStatus = (NotSending, SendingData, SendingCtl);
VAR
        state12, state21 : provStatus;
        delay12, delay21 : INTEGER;
INITIALIZE
    BEGIN
        state12 := NotSending;
        state21 := NotSending;
    END;

TRANS
PROVIDED state12=NotSending
        WHEN n[1].RR
            BEGIN
                {$R95} OUTPUT n[2].RR(send_seq,recv_seq);
                delay12 := PRANDOM(1,1,10);
                state12 := SendingCtl;
            END;
        WHEN n[1].REJ
            BEGIN
                {$R95} OUTPUT n[2].REJ(send_seq,recv_seq);
                delay12 := PRANDOM(1,1,10);
                state12 := SendingCtl;
            END;
        WHEN n[1].I
            BEGIN
                IF (random(100)<5) THEN ok:=FALSE; { corrupt }
                {$R90} OUTPUT n[2].I(send_seq,recv_seq,data,ok);
                delay12 := PRANDOM(1,4,10);
                state12 := SendingData;
            END;
PROVIDED state12=SendingCtl
    DELAY (delay12)
        BEGIN
            OUTPUT pp[1].provReady;
            state12 := NotSending;
         END;
PROVIDED state12=SendingData
    DELAY (delay12)
        BEGIN
            OUTPUT pp[1].provReady;
            state12 := NotSending;
         END;
TRANS
PROVIDED state21=NotSending
        WHEN n[2].RR
            BEGIN
                {$R95} OUTPUT n[1].RR(send_seq,recv_seq);
                delay21 := PRANDOM(1,1,10);
                state21 := SendingCtl;
            END;
        WHEN n[2].REJ
            BEGIN
                {$R95} OUTPUT n[1].REJ(send_seq,recv_seq);
                delay21 := PRANDOM(1,1,10);
                state21 := SendingCtl;
            END;
        WHEN n[2].I
            BEGIN
                IF (random(100)<5) THEN ok:=FALSE; { corrupt }
                {$R90} OUTPUT n[1].I(send_seq,recv_seq,data,ok);
                delay21 := PRANDOM(1,4,10);
                state21 := SendingData;
        END;
PROVIDED state21=SendingCtl
    DELAY (delay21)
        BEGIN
            OUTPUT pp[2].provReady;
            state21 := NotSending;
         END;
PROVIDED state21=SendingData
    DELAY (delay21)
        BEGIN
            OUTPUT pp[2].provReady;
            state21 := NotSending;
         END;
END;

{**********************************************************************

        THE SIMPLE LINK ACCESS PROCEDURE/BALANCED PROTOCOL

**********************************************************************}

MODULE SLAP_type SYSTEMPROCESS(cep_id : cep_type);
        IP  a: user_slap_chan(slap_role) INDIVIDUAL QUEUE;
            n: slap_net_chan(slap_role) INDIVIDUAL QUEUE;
            pp: ProvState_chan(slap_role) INDIVIDUAL QUEUE;
END;

BODY SLAP_body FOR SLAP_type;
CONST
        win_top = 1; { win_size-1 }
        msg_buf_size = 20;
```

```
TYPE
        ClockState = (Running, Stopped, Expired);
VAR
        sv, rv:INTEGER; { Send/receive sequence numbers }
        Clock: ClockState;
        send_win_in, send_win_out, send_win_now: INTEGER;
        unsent_count, send_count, lastACK, ACKSdue:INTEGER;
        send_win: ARRAY[0..win_top] OF data_type;
        sent_frame, provAvail: BOOLEAN;
STATE
        NRML, REJ;

{******************************************************************
        UTILITY ROUTINES
*******************************************************************}

{ l is a debugging routine, used to print useful info and distinguish
  the two sides by printing on the left or right of the page }

PROCEDURE l;
BEGIN
    IF cep_id=2 THEN WRITE('                       ');
    WRITELN('s_win_in=',send_win_in,' s_win_out=',send_win_out,'
                s_count=',send_count);
    IF cep_id=2 THEN WRITE('                       ');
    WRITELN('unsent_count=',unsent_count,' sv=',sv,' rv=',rv);
    IF cep_id=2 THEN WRITE('               ');
END;

FUNCTION process_ACK(send_seq, recv_seq:INTEGER):INTEGER;
VAR
        ACK_count:INTEGER;
BEGIN
        { Determine how many frames were acknowledged }

        ACK_count := recv_seq-lastACK;
        IF ACK_count<0 THEN ACK_count := ACK_count+8;
        process_ACK := ACK_count;

        { Move forward in the send window }

        l; WRITELN(ACK_count,' frames acknowledged');
        IF (ACK_count>0) AND (ACK_count<=ACKSdue) THEN BEGIN
            lastACK := recv_seq;
            send_count := send_count - ACK_count;
            send_win_out := (send_win_out + ACK_count) MOD win_size;
            ACKSdue := ACKSdue - ACK_count;
            Clock := Stopped;
        END;
END;

PROCEDURE resend(send_seq, recv_seq:INTEGER);
VAR
        resend_count:INTEGER;
BEGIN
        resend_count := sv-recv_seq;
        IF resend_count<0 THEN resend_count:=resend_count+8;
        IF (resend_count>0) AND (resend_count<=win_size) THEN BEGIN
            l; WRITELN('Resending ',resend_count,' frames');
            unsent_count := unsent_count+resend_count;
            sv := recv_seq;
            send_win_now := send_win_now - resend_count;
            ACKSdue := ACKSdue-resend_count;
            IF (send_win_now<0)
                    THEN send_win_now := send_win_now + win_size;
        END ELSE BEGIN
                l; WRITELN('resend count zero; sv=',sv,'  recv_seq=',recv_seq);
            END;
END;
```

```
PROCEDURE process_NAK(send_seq:INTEGER; recv_seq:INTEGER);
BEGIN
        resend(send_seq, recv_seq);
END;

PROCEDURE update_info;  { Called after sending an I frame }
BEGIN
        l;WRITELN('Sent I(',sv,',',rv,')');
        sv := (sv+1) MOD 8;
        send_win_now := (send_win_now + 1) MOD win_size;
        unsent_count := unsent_count-1;
        ACKSdue := ACKSdue + 1;
END;

{*********** INITIALIZATION **************************}

INITIALIZE
    TO NRML
        BEGIN
            lastACK := 0;
            ACKSdue := 0;
            rv := 0;
            sv := 0;
            send_count := 0;   { Number of frames in window }
            unsent_count := 0; { Number of frames to send   }
            send_win_in := 0;
            send_win_out := 0;
            send_win_now := 0;
            Clock := Stopped;
            provAvail := TRUE;
            sent_frame := FALSE;
        END;

{******************************************************************
        PROVIDER BECOMES AVAILABLE
*******************************************************************}

TRANS
    WHEN pp.provReady
        BEGIN
            IF (ACKSdue > 0) AND (Clock = Stopped) THEN BEGIN
                l;WRITELN('Started clock');
                Clock := Running;
            END;
            provAvail := TRUE;
            l;WRITELN('Channel cleared');
        END;

{******************************************************************
        TIMEOUT
*******************************************************************}

TRANS
    PROVIDED Clock=Running
        DELAY (Timeout)
            BEGIN
                Clock := Expired;
            END;

{************************************************************
        SERVICE FOR UPPER LAYER
*************************************************************}

TRANS
PRIORITY 30     { Second lowest priority }
    WHEN a.send_request
        BEGIN
            send_win[send_win_in] := data;
            send_win_in := (send_win_in+1) MOD win_size;
            send_count := send_count+1;
            unsent_count := unsent_count+1;
            l; WRITELN('Got send_req ',send_win_in);
        END;

{****************************************************************************
        UNSOLICITED INFORMATION TRANSFER
****************************************************************************}

TRANS
PRIORITY 40     { Unsolicited data transfer is lowest priority }
    FROM NRML
    PROVIDED (unsent_count>0) AND provAvail
        BEGIN
            OUTPUT n.I(sv,rv,send_win[send_win_now],TRUE);
            provAvail := FALSE;
            update_info;
        END;

{***********************************************************************
        HANDLING OF IN SEQUENCE I-FRAMES
***********************************************************************}

TRANS
    FROM NRML
    PRIORITY 30
        WHEN n.I
            { Got I frame to send, received I frame in sequence }
            PROVIDED (unsent_count>0) AND (rv=send_seq) AND (ok=TRUE) AND provAvail
                BEGIN
                    OUTPUT a.receive_response;
                    l;WRITELN('Dequeue I(',send_seq,',',recv_seq,')');
                    IF (process_ACK(send_seq,recv_seq)>0) THEN
                        sent_frame := TRUE;
                    rv := (rv+1) MOD 8;
                    OUTPUT n.I(sv,rv,send_win[send_win_now],TRUE);
                    update_info;
                    provAvail := FALSE;
                END;

{$T ============ GROUP SPLIT HERE ===================}

            { No I frame to send, received I frame in sequence }
            PROVIDED (unsent_count=0) AND (rv=send_seq) AND (ok=TRUE) AND provAvail
                BEGIN
                    OUTPUT a.receive_response;
                    l;WRITELN('Dequeue I(',send_seq,',',recv_seq,')');
                    IF (process_ACK(send_seq,recv_seq)>0) THEN
                        sent_frame := TRUE;
                    rv := (rv+1) MOD 8;
                    OUTPUT n.RR(sv,rv);
                    l; WRITELN('14 Sent RR(',sv,',',rv,')');
                    provAvail := FALSE;
                END;

{*********************************************************************
        HANDLING OF OUT-OF-SEQUENCE/CORRUPT I-FRAMES
*********************************************************************}

{ When an out-of-sequence/corrupt frame is received, we go into the REJ state.
  We trash any incoming I frames on the data channel, and then send a REJ
  requesting retransmission. }

TRANS
    PRIORITY 30
        FROM NRML TO REJ
            WHEN n.I
                PROVIDED (rv<>send_seq) OR (ok=FALSE)
                    BEGIN
                        l;IF ok THEN WRITELN('Dequeued out of sequence I(',
                                       send_seq,',',recv_seq,') expected ',rv)
                        ELSE WRITELN('Dequeued corrupt I(',send_seq,',',
                                     recv_seq,') expected ',rv);
                    END;
TRANS
        PRIORITY 10
        FROM REJ TO REJ
            WHEN n.I
                BEGIN
                    l;WRITELN('Trashed I(',send_seq,',',recv_seq,')');
                END; { dequeue I frames }

TRANS
        PRIORITY 20
        FROM REJ TO NRML
                PROVIDED provAvail
            BEGIN
                OUTPUT n.REJ(sv,rv);
                l; WRITELN('Sent REJ(',sv,',',rv,')');
                provAvail := FALSE;
            END;

{**************************************************************
        HANDLING OF OTHER PROVIDER EVENTS
**************************************************************}

TRANS
    { Response to physical layer events }
    FROM NRML
    PRIORITY 20
            WHEN n.RR
                BEGIN
                    l;WRITELN('Dequeue RR(',send_seq,',',recv_seq,')');
                    IF (process_ACK(send_seq,recv_seq)>0) THEN
                        sent_frame := TRUE;
                END;
            WHEN n.REJ
                BEGIN
                    l;WRITELN('Dequeue REJ(',send_seq,',',recv_seq,')');
                    IF (process_ACK(send_seq,recv_seq)>0) THEN
                        sent_frame := TRUE;
                    process_NAK(send_seq,recv_seq);
                END;

{*******************************************************************
        SEND INDICATION
*******************************************************************}

TRANS
    PROVIDED sent_frame
        BEGIN
            OUTPUT a.send_indication;
            sent_frame := FALSE;
        END;

{*******************************************************************
        HANDLE TIMEOUT
*******************************************************************}

{ We treat the timeout as a REJ, and resend last I frame if there is one }

TRANS
    PRIORITY 15
        PROVIDED (Clock=Expired)
            BEGIN
                IF (ACKSdue>0) THEN BEGIN
                    l;WRITELN('TIMED OUT!');
                    IF sv=0 THEN process_NAK(rv,7)
                    ELSE process_NAK(rv,sv-1);
                END;
                Clock := Stopped; { stop until resend takes place }
            END;

END; { of Protocol body }

{*******************************************************************

        USER MODULE BODY

********************************************************************}


MODULE user_type SYSTEMPROCESS(cep_id: cep_type);
        IP a: user_slap_chan(user_role) INDIVIDUAL QUEUE;
END;

BODY user_body FOR user_type;
CONST
        MinPollDelay = 5;
        MaxPollDelay = 40;
        MinSendDelay = 5;
        MaxSendDelay = 40;
VAR
        credit : INTEGER;

INITIALIZE
        BEGIN
            credit := 2;
        END;

TRANS { Dequeue and discard incoming frames }
        WHEN a.receive_response
            BEGIN
                IF (cep_id=2) THEN WRITE('                    ');
                WRITELN('User discards receive_response');
            END;

TRANS { Occasionally send data }
{       DELAY (MinSendDelay,MaxSendDelay)
        As the above line is commented out, we always
        send another frame immediately we have credit. }
            PROVIDED credit>0
            BEGIN
                OUTPUT a.send_request(random(80));
                credit := credit - 1;
                IF (cep_id=2) THEN WRITE('                    ');
                WRITELN('User outputs send request, credit ',credit);
            END;

TRANS { Message was sent successfully; get credit }
        WHEN a.send_indication
            BEGIN
                credit := credit + 1;
                IF (cep_id=2) THEN WRITE('                    ');
                WRITELN('User got send indication, credit ',credit);
            END;
END;

{*******************************************************************

        SPECIFICATION BODY

********************************************************************}

MODVAR
        app: ARRAY[cep_type] OF user_type;
        SLAP: ARRAY[cep_type] OF SLAP_type;
        provider: provider_type;

INITIALIZE
        VAR
```

```
            child: cep_type;
        BEGIN
            {$O1 Disable WRITEs}
            INIT provider WITH provider_body;
            FOR child:=ep_low TO ep_high DO BEGIN
                INIT app[child] WITH user_body(child);
                INIT SLAP[child] WITH SLAP_body(child);
                CONNECT app[child].a TO SLAP[child].a;
                CONNECT SLAP[child].n TO provider.n[child];
                CONNECT SLAP[child].pp TO provider.pp[child];
            END;
        END;
END.
```

Appendix D

build2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

main() {
    char cmd[20];
    FILE *fp;
    int timeout, i;
    for (timeout=40; timeout<=40;timeout+=10) {
        FILE *fp = fopen("mat.txt","w");
        float p, q, r, t = (float)timeout;
        fprintf(fp,"5\n"); /* 5x5 matrix */
        /* row 1 */
        p = .1+.0035*t;
        q = .32-0.0022*t;
        r = 0.19-.0013*t;
        fprintf(fp,"%.4f 0.0700 %.4f 0.3200 %.4f    %3f %3f 1.   0.025   0.\n",
            p,q,r,t/5.,t*3./5.);
        /* row 2 */
        p = 0.74+0.00365*t;
        fprintf(fp,"%.4f 0.0000 0.0000 %.4f 0.000    0.5 0. 0. 0. 0.\n",
            1.-p,p);
        /* row 3 */
        fprintf(fp,"0.85 0.0000 0.0000 0.15 0.000    2.8 0. 0. 0. 0.\n");
        /* row 4 */
        p = .25+.00325*t;
        fprintf(fp,"%.4f 0.0000 %.4f 0. 0.    4. 0. 0.25 0. 0.\n",1.-p,p);
        /* row 5 */
        fprintf(fp,"0.426 0. 0.574 0. 0.    3.5 0. 1.3 0. 0.\n");
        fprintf(fp,"\nTIMEOUT %d\n\n",timeout);
        fclose(fp);
        sprintf(cmd,"pa -A model.pre");
        system(cmd);
    }
}
```

Bibliography

[12590] Esprit Project 125. "GRASPIN - Towards the Next CASE Generation". Technical Paper GRA 125/2, Gesellschaft für Mathematik und Datenverarbeitung mbH, March 1990.

[ABG90] A. V. Aho, B. S. Bosik, and S. J. Griesmer. "Protocol Testing and Verification within AT&T". *AT&T Technical Journal*, 69(1):4–6, January/February 1990.

[ABM88] S. Aggarwal, D. Barbara, and K. Meth. "A Software Environment for the Specification and Analysis of Problems of Coordination and Concurrency". *IEEE Transactions on Software Engineering*, 14(3):280–290, March 1988.

[Bal85] R. Balzer. "A 15 Year Perspective on Automatic Programming". *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[BB87] T. Bolognesi and E. Brinksma. "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems*, (14):25–59, 1987.

[BGS87] G. Bochmann, G. Gerber, and J. Serre. "Semi-automatic Implementation of Communication Protocols". *IEEE Transactions on Software Engineering*, SE-13(9):989–1000, September 1987.

[BH85] P. Brinch Hansen. *"Brinch Hansen on Pascal Compilers"*. Prentice-Hall, 1985.

[BK84] M. Booyens and P.S. Kritzinger. "Snap/L: A Language to describe and evaluate queueing network models". *Performance Evaluation*, 4(3):171–182, August 1984.

[BMW89] H. Beilner, J. Mäter, and N. Weissenberg. "Towards a Performance Modelling Environment: News on HIT". In *Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma, Spain, 1988*. Plenum, 1989.

[Boc90] G. V. Bochmann. "Specifications of a Simplified Transport Protocol using different Formal Description Techniques". *Computer Networks and ISDN Systems*, (18):335–377, 1989/90.

[BRW90] M. Bush, K. Rasmussen, and F. Wong. "Conformance Testing Methodologies for OSI Protocols". *AT&T Technical Journal*, 69(1):84–100, January/February 1990.

[CA91] S. C. Chamberlain and P. D. Amer. "Broadcast Channels in Estelle". *IEEE Transactions on Computers*, 40(4):423–436, April 1991.

[CLH90] C. G. Cassandras, J. I. Lee, and Y. C. Ho. "Efficient Parametric Analysis of Performance Measures of Communication Networks". *IEEE Journal on Selected Areas in Communications*, 8(9):1709–1722, December 1990.

[Con88] A. E. Conway. "A Generic Performance Model for Multi-Layered OSI Communication Architectures". Technical Memorandum TM-0069-10-88-414.17, GTE Laboratories, October 1988.

[Con91] A. E. Conway. "A Perspective on the Analytical Performance Evaluation of Multilayered Communication Protocol Architectures". *IEEE Journal on Selected Areas in Communications*, 9(1):4–14, January 1991.

[Cou87] J.P. Courtiat. "How could Estelle become a better FDT?". In H. Rudin and C.H. West, editors, *Protocol Specification, Testing and Verification 7*. Elsevier Science Publishers B.V. (North-Holland), 1987.

[CS88] C. G. Cassandras and S. G. Strickland. "Perturbation Analytic Methodologies for Design and Optimization of Communication Networks". *IEEE Journal on Selected Areas in Communications*, 6(1):158–171, January 1988.

[DAC$^+$] M. Diaz, J.-P. Ansart, J.-P. Courtiat, P.Azema, and V.Chari, editors. *"The Formal Description Technique Estelle: Results of the ESPRIT SEDOS Project)"*. Elsevier Science Publishers B.V. (North-Holland).

[EWS] "EWS - An Integrated Workstation for the Design of Distributed Software". Promotional literature, SEDOS Consortium.

[Fle88] A. Fleischmann. "PASS - The Parallel Activity Specification Scheme". Technical Report 43.8715, IBM European Networking Center, 1988.

[Goo88] R. Goodman. *"Introduction to Stochastic Models"*. Benjamin Cumings, 1988.

[GR91] P. Gburzyński and P. Rudnicki. "LANSF: A Protocol Modelling Environment and its Implemenation". *Software: Practice and Experience*, 21(1):51–76, January 1991.

[HEC79] Y. C. Ho, M. A. Eyler, and T. T. Chien. "A Gradient Technique for General Buffer Storage Design in a Serial Production Line". *International Journal of Production Research*, 17(6):557–580, 1979.

[Ho87] Y. C. Ho. "Performance Evaluation and Perturbation Analysis of Discrete Event Dynamic Systems". *IEEE Transactions on Automatic Control*, AC-32(7):563–572, July 1987.

[Hoa85] C. Hoare. *"Communicating Sequential Processes"*. Prentice-Hall, 1985.

[Hol90] G. L. Holzmann. "Algorithms for Automated Protocol Verification". *AT&T Technical Journal*, 69(1):32–44, January/February 1990.

[Hol91] G. Holzmann. *"The Design and Validation of Computer Protocols"*. Prentice-Hall, 1991.

[How71] R.A. Howard. *"Dynamic Probabilistic Systems. Volume I: Markov Models. Volume II: Semi-Markov and Decision Processes"*. John Wiley and Sons, 1971.

[ISO89] *"ISO IS 9074, Information Processing Systems - Open Systems Interconnection - Estelle - A Formal Description Technique based on an Extended State Transition Model"*, 1989.

[JdSS92] J.P.Courtiat and P. de Saqui-Sannes. "ESTIM: an Integrated Environment for the Simulation and Verification of OSI Protocols specified in Estelle". *Computer Networks and ISDN Systems*, (25):83–98, 1992.

[JMG88] C. JARD, J-F Monin, and R Groz. "Development of Veda, a Prototyping Tool for Distributed Algorithms". *IEEE Transactions on Software Engineering*, 14(3):339–352, March 1988.

[Kle76] L. Kleinrock. *"Queueing Systems (Volume 1)"*. John Wiley and Sons, 1976.

[Kle88] L. Kleinrock. "Performance Evaluation of Distributed Computer- Communication Systems". In O. J. Boxma and R. Syski, editors, *Queueing Theory and its Applications*, pages 1–57. Elsevier Science Publishers, 1988.

[Kri86] P. S. Kritzinger. "A Performance Model of the OSI Communication Architecture". *IEEE Transactions on Communications*, 34(6):554–563, 1986.

[Kri89] P. S. Kritzinger. "Theoretical Modelling as an Aid to Protocol Implementation". Technical Report CS-89-01-00, Department of Computer Science, University of Cape Town, January 1989.

[KvD88] P. S. Kritzinger and J. van Dijk. "An Estelle Compiler for a Protocol Development Environment". Technical Report CS88-02-00, Department of Computer Science, University of Cape Town, June 1988.

[KW90] P. S. Kritzinger and G. Wheeler. "A Protocol Engineering Workstation". In Son T. Vuong, editor, *Formal Description Techniques II, Proceedings of the IFIP TC/WG 6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, Forte '89, Vancouver, Canada, 5–8 December 1989*, pages 53–59. Elsevier Science Publishers B.V. (North-Holland), 1990.

[KW91] P. S. Kritzinger and G. Wheeler. "Automated Protocol Performance Analysis". Technical Report CS91-02-00, Department of Computer Science, University of Cape Town, October 1991.

[KW93] P.S. Kritzinger and G. Wheeler. "Semi-Markovian Analysis of Protocol Performance". In G. Leduc A. Danthine and P. Wolper, editors, *Protocol Specification, Testing and Verification 13*. Elsevier Science Publishers B.V. (North-Holland), 1993.

[Lav88] S. S. Lavenberg. "A Perspective on Queueing Models of Computer Performance". In O. J. Boxma and R. Syski, editors, *Queueing Theory and its Applications*, pages 59–94. Elsevier Science Publishers, 1988.

[Lin90] R. J. Linn. "Conformance Testing for OSI Protocols". *Computer Networks and ISDN Systems*, (18):203–220, 1989/90.

[LS84] S.S. Lam and A.U. Shankar. "Protocol Verification via Projections". *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[MIL89] MIL 3, Inc. "OPNET - Optimised Network Engineering Tools - M Version - Technical Overview". Technical report, MIL 3, Inc, 1989.

[Nas87] S. C. Nash. "Format and Protocol Language (FAPL)". *Computer Networks and ISDN Systems*, (14):61–77, 1987.

[New91] D. New. "protocol visualisation". Technical Report 91-18, Department of Computer and Information Sciences, University of Delaware, May 1991.

[Peh90] B. Pehrson. "Protocol Verification for OSI". *Computer Networks and ISDN Systems*, (18):185–202, 1989/90.

[Ray87] D. Rayner. "OSI Conformance Testing". *Computer Networks and ISDN Systems*, (14):79–98, 1987.

[RH80] C. V. Ramamoorthy and G. S. Ho. "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets". *IEEE Transactions on Software Engineering*, SE-6(5):440–449, September 1980.

[RL80] M. Reiser and S.S. Lavenberg. "Mean Value Analysis of Closed Multiclass Queueing Networks". *Journal of the ACM*, 27(2):313–322, April 1980.

[Ros90] M. T. Rose. *"The Open Book - A Practical Perspective on OSI"*. Prentice-Hall, 1990.

[RS82] A. Rockström and R. Saracco. "SDL – CCITT Specification and Description Language". *IEEE Transactions on Communications*, 30(6):1310–1318, June 1982.

[Rub89] R. Y. Rubinstein. "Sensitivity Analysis and Performance Extrapolation for Computer Simulation Models". *Operations Research*, 37(1):72–81, January-February 1989.

[Rud83] H. Rudin. "From Formal Protocol Specification towards Automated Performance Prediction". In H. Rudin and C. H. West, editors, *Protocol Specification, Testing and Verification 3.* Elsevier Science Publishers B.V. (North-Holland), 1983.

[Rud85] H. Rudin. "An Improved Algorithm for Estimating Protocol Performance". In Y. Yemini, R. Strom, and S. Yemini, editors, *Protocol Specification, Testing and Verification 5.* Elsevier Science Publishers B.V. (North-Holland), 1985.

[SAP] SAPONET. "SAPONET-P – The Switched Data Network of the South African Post Office – X.25 Packet Switching". Technical publication no. 1, issue 3, The Director, Telematic Services.

[SB90] D. P. Sidhu and T. P. Blumer. "Semi-automatic Implementation of OSI Protocols". *Computer Networks and ISDN Systems*, (18):221–238, 1989/90.

[SC81] C. H. Sauer and K. Mani Chandy. *"Computer Systems Performance Modelling"*. Prentice-Hall, 1981.

[SG78] G. Scott Graham. "Queueing Networks Models of Computer System Performance". *Computing Surveys*, 10(3):219–224, September 1978.

[SG89] R. Sijelmassi and P. Gaudette. "An Object-Oriented Model for Estelle". In K. Turner, editor, *Formal Descriptuion Techniques I.* North-Holland, 1989.

[SMC90] M. Sczittnick and B. Muller-Clostermann. "MACOM - A Tool for the Markovian Analysis of Communication Systems". In *Proceedings of the 4th International Conference on Data Communication Systems and their Performance, Barcelona, Spain, June 20–22, 1990*, 1990.

[Sun78] C. A. Sunshine. "Survey of Protocol Definition and Verification Techniques". *Computer Networks*, (2):346–350, 1978.

[Sur89] R. Suri. "Perturbation Analysis: The State of the Art and Research Issues Explained via the G/G/1 Queue". *Proceedings of the IEEE*, 77(1):114–137, January 1989.

[Tah89] H. A. Taha. *"Operations Reasearch - An Introduction (4th ed.)"*. Maxwell Macmillan, 1989.

[vB90] G. v. Bochmann. "Protocol Verification for OSI". *Computer Networks and ISDN Systems*, (18):167–184, 1989/90.

[Whe90a] G. Wheeler. "A Software Development Environment for the Evaluation of Computer Protocols". Technical Report CS-89-03-01, Department of Computer Science, University of Cape Town, January 1990.

[Whe90b] G. Wheeler. “Specification of the Estelle E-Code”. Unpublished internal document, 1990.

[Whe90c] G. Wheeler. *“The Estelle PEW User Manual”*, 1990.

[Whe91] G. Wheeler. “Protocol Engineering from Formal Specifications”. Ph.D. Thesis, Department of Computer Science, University of Cape Town, October 1991.