Thesis:

PETRI NET MODELLING OF A COMMUNICATIONS PROTOCOL

Author:

Colin J. McAllister B.Sc.

Supervisor:

Dr. Michael Scott Ph.D.

Submitted to:

The Dublin City University

School of Computer Applications

For the Degree of Master of Science.

Date: 28th July 1989

ACKNOWLEDGEMENT

I wish to thank Dr. Michael Scott for his helpful direction during this project.

DECLARATION

This dissertation is based on the author's own work. It has not previously been submitted for a degree at any academic institution.

Colin McAllister

Colin McAllister

28th July 1989.

Thesis:

PETRI NET MODELLING OF A COMMUNICATIONS PROTOCOL

Author: Colin McAllister B.Sc.

Supervisor: Dr. Michael Scott Ph.D.

ABSTRACT

The Petri net is a formal modelling tool applicable to distributed systems and communication protocols. Two methods of analysis are applied to formal models of the "Alternating Bit Protocol".

(i) A timed Petri net model is simulated to measure protocol performance.

(ii) A modular numeric Petri net model is validated by reachability analysis.

The simulation and validation tools are programmed in (i) "C" language and (ii) Prolog. A specification language "Needle" is developed. It describes the model system as a hierarchy of modular state transition networks. The model is searched for all possible event sequences, and the result displayed as a reachability tree. The specification language is capable of describing models which execute backwards in simulation time. The modular numeric Petri net is the basis of a powerful computer architecture, capable of parsing its own specification language to build complex models. Attention is drawn to the similarities between Petri net theory and quantum mechanics.

KEY WORDS: Petri Nets, Prolog, Formal Specification, Protocol Validation, Timed Simulation.

1 TABLE OF CONTENTS

APPENDICES

1 INTRODUCTION

1.1 Formal Methods For Communications Software

This research project examined formal methods for the development of data communications software. The techniques studied apply generally to the analysis of complex systems.

Communications software is particularly complex for several reasons:

- It executes concurrently at multiple locations and on different levels of hardware. E.g. host computer, front end processor and packet switching exchange.
- It must perform at sufficient speed to interact with the communications hardware.
- It must communicate data reliably, despite noisy transmission channels and intermittent equipment failure.
- It is designed as separate functional layers, and possibly by separate companies. The interfaces between layers must be specified and tested.

The software algorithms used for transmitting and receiving data are referred to as a protocol. The "Alternating Bit Protocol" is a simple example, which can communicate data reliably over a noisy transmission channel.

Application of validation methods to a complex design can produce significant savings in project development costs, by high-lighting design difficulties at an early stage. This thesis presents two complementary methods of analysing Petri net models of the "Alternating Bit Protocol".

1.2 Two Methods of Protocol Analysis

This project examines two methods of protocol analysis: "Performance Modelling" by timed simulation and "Formal Validation" by reachability analysis. A specification language is developed for each method. The methods use a Petri net model which is extended to represent:

timed events,

numeric operations,

and modular structure.

Following the introduction (chapter 1), this paper contains two sections, Method-I and Method-II, describing distinctly different approaches. Finally, chapter 11 discusses potential developments from Petri net theory.

Method-**I** Performance Modelling:

Analyses the performance of the protocol by timed simulation. A simulation tool "SIM" is developed in the "C" language. The method is based on a timed Petri net model. A simulation language builds timed models from standard components. (Chapters 2 to 4, appendices A to C).

Method-**II** Formal Validation:

A formal specification of the protocol is validated by reachability analysis. A validation tool "TRAV" is developed in "Prolog". The method is based on a modular numeric Petri net model, for which a formal language "Needle" is developed. (Chapters 5 to 10, appendices D to J).

1.3 Definition of Petri Net and Reachability Tree

The standard Petri net model is a directed graph containing **places, transitions** and directed **arcs.** These are drawn as circles, bars and connecting arrows. The Petri net executes by moving **tokens** from place to place. Each movement of tokens is a discrete event corresponding to the firing of a transition.

A Petri net marking **M** is an instantaneous state of the model, characterised by the number of tokens at each place. A marking **Mn** is said to be reachable from an initial marking **M0,** if their exists a possible firing sequence **M0, M1... Mn** by which marking **Mn** can be reached. A reachability tree is a structured representation of all possible markings reachable from **M0.** Every possible firing sequence is represented by a path though the tree, from its root **M0.** Leaf nodes of the tree are either duplicate markings or deadlock markings. A duplicate marking occurs when a firing sequence loops to a previous state of the model. A deadlock marking is a state of the model in which no further events are possible.

Reachability analysis is an automatic method of generating a reachability tree from the specification of a Petri net.

[AJM086] for instance, gives a good description of Petri nets and reachability trees.

1.4 Analysis Method-I: Performance Modelling

This method uses a Timed Petri Net model (TPN), to simulate protocol performance. Data corruption on a noisy communications channel is simulated, and the effect on transmission rate measured.

The simulation tool "SIM" is developed in "C". The execution speed of a concise "C" program allows fast iteration through long pseudo-random simulation runs. The results of the simulation are presented as a performance graph of the protocol.

1.5 Analysis Method-II: Formal Validation

This method examines a modular Numeric Petri Net model (NPN). A structured specification language "Needle" is developed to describe a complex system as a hierarchy of modular state-transition networks. The validation tool "TRAV" is developed in Prolog. The inherent backtracking search of Prolog traverses all possible execution sequences of the model system. The search result is reported as a reachability tree, with identification of deadlock and looping conditions. Prolog was found to be an ideal language for developing a validation tool. Among its useful features are:

declarative programming,

backtracking execution,

conciseness in manipulating lists,

database facilities.

1.6 The Petri Net Analysis Tools

The research resulted in development of two analysis tools. Both tools process the model specification as a text file, and print analysis reports. The simulation tool "SIM" provides an animated display of model execution. The validation tool "TRAV" provides menu driven control of validation experiments. Both tools run on an IBM personal computer.

2 VARIOUS APPROACHES TO TIMED SIMULATION

2.1 Analysis Tools for Discrete Event Systems

An international epidemic of analysis tools for "Discrete Event Systems" is now emerging from computer research establishments. Over 20 of these [FELD86, BILL88] have the Petri net model as their mathematical foundation. When analysing the efficiency of systems, it becomes essential to analyse timing properties. Petri net models, extended to represent time, have found their niche [MOLL89]. [COHE89] conclude that the mathematical theory of discrete event systems is still in its infancy.

The favourite areas of application are manufacturing models [FAN_88, GERS89] and communication protocols [PETR66, BILL88], but this will broaden as the technology becomes established. The Petri net is a completely general model, with potential applications ranging from fundamental particle theory [HASS89], to satellite control systems [CIAR87].

2.2 Representations of Time

There are different approaches to modelling time. The simulation method represents time as a real number **T** which increments in measured amounts. Measurement is necessary to make observations about the performance efficiency of a protocol.

The "Temporal Logic" approach [SANT88], which reasons logically about the passage of time, is suitable for verifying that a protocol is correct, but not for quantitative performance assessment.

2.3 Basic and Complex Petri Net models

Petri net theory, originated in 1962 by C.A. Petri [PETR62,PETR66] now has a bibliography of over 2000 articles [BILL88, ROZE87].

The Petri net model is a directed graph with two types of node. In its simplest form the graph is specified by a 2-dimensional incidence matrix. Execution of the model is a simple matrix operation on a 1-dimensional state vector. Many analysis techniques are available.

At greater complexity, modelling concepts related to the Petri net are specified by powerful visual formalisms [HARE86, KARA88, SCHI88], and executed as hierarchical multiprocess computations [AGHA86, ARCH87, SEVI88].

2.4 Petri Nets Mapped onto Concurrent Processes

The Petri net is a model with very fine grained concurrency, the model is composed of many small processing units, all in activity at the same time. Contrast this with the sequential architecture of a microcomputer, at any instant only one single "byte" of memory (RAM) is active in the whole address space of the processor (CPU).

In this project, I have taken the approach of mapping the Petri net model onto a concurrent process model. The application is specified as a Petri net model. Concurrent processing is the internal model of the simulation tool. The tool is written in 'C', a sequential language for a single processor, so the concurrency is just a model, and not physical.

A concurrent process model is also the basis of [BERZ88] on "Rapidly Prototyping Real-Time Systems". Their modelling tool is part of an integrated development environment. Their specification language PSDL describes a directed graph with timing and control constraints. They emphasis the importance of a unified representation of data flow and control flow. This is a natural property of the Petri net, which has only one type of directed arc for representing all classes of flow. [BERZ88] benefits from previous work on Petri nets, of which [BRUN86] is relevant here, "Process Translatable Petri Nets for the Rapid Prototyping of Process Control Systems".

2.5 Integration of Petri and Queueing Models

Queuing models have long been used to analyse the performance of data processing systems. But they are inadequate to express the synchronisation requirements of concurrent systems. [CHAN89] presents a combined approach in a modelling tool TPQN or "Timed Petri/Queuing Network". A textual specification language TPQL allows a model to be built from the interconnection of component places, queues and transitions. Simulation is applied to a model of an operating system scheduler, and performance graphs are generated. This is very similar to my approach (Chapter 3), a structured classification of standard Petri net components is provided in Appendix-A.

2.6 Stochastic Petri Net Analysis with 'C'.

Stochastic analysis of Petri nets is a well established method of performance analysis, based on the Markov chain [AJM086]. A stochastic Petri net is a timed model with exponentially distributed firing times. [DUGA89] use the method to develop voting algorithms for a distributed file system. The performance model is sufficently detailed to simulate the failure and repair of host computers. The Stochastic Petri net specification language is based on the 'C' language. The specification is written as a 'C' program and calls a set of predefined functions e.g. place(), transition(), output_arc(), input_arc().

2.7 Discrete Event Simulation with Prolog

The "Chameleon" simulator [FAN_88] is built in the Prolog language. It models time based discrete event simulation. The system to be modeled, in this case a manufacturing system, is broken up into logical modules, and encoded in Prolog. The simulator regulates the occurance of events and drives an animated display of the system state. Their project explains some of the practical difficulties, e.g. running Prolog on a personal computer with only 640 kilobytes of memory. They stress the importance of modular design, and use a system of replaceable modular units which communicate via queues.

2.8 Temporal Extension to a Specification Language

Another approach to timed modelling is to take an existing specification language and extend it with temporal logic [NIX089]. A functional specification language "Ina Jo" has its assertion language enriched with a branching time temporal logic system. The extension method was carefully chosen, and fitted in with the state transition model on which the specification language was based. A limitation of the language is the absence of support for modular programming.

2.9 Development of the "SIM" Simulation Tool

A simulator "SIM" was developed [MCAL87] for performance analysis of distributed systems and communication protocols. The model system is expressed as a timed Petri net model. Simulation of the model confirms its correct behavior, and measures its performance. The tool has been tested on a number of small case studies, including the alternating bit communications protocol, manufacturing work flow and an alarm system.

3 THE TIMED PETRI NET SIMULATION TOOL

3.1 Brief Description of the Simulation Tool

A simulation tool "SIM" was developed to carry out performance analysis. It simulates a Petri net model, extended to represent time. The Petri net is appropriate for analysing any system of component parts which interact in a logically defined manner.

The tool has a specification language for defining application models. A specification is a formal statement of a model, including its time dependent properties. The simulator executes the specification, displaying the sequence of events.

The specification language controls automated collection of performance statistics. The results can be presented as a performance graph using a standard spreadsheet program. The simulation uses randomised numbers, so accuracy of results depends on the number of program runs.

The simulation tool is programmed in 'C' and has a modular structure, allowing it to be extended with new features.

3.2 Timed Petri Net Specification Language

The specification language defines application models as timed Petri nets. The specification describes a network of named places and transitions. The connecting arcs are described by the key words **FROM** and **TO** which identify the source and destination places of every transition. The language calls up a standard place type for each simulation function required; queues, timers etc. Parameter values

specify the place characteristics such as time delays, and event probabilities. Customised collection and reporting of statistics is also controlled via the specification language.

I have not formalised the language, but its syntax can be seen from the example specifications (Appendix-B) and [MCAL87].

[CHAN89] define an almost identical language for the same purpose, performance analysis of Timed Petri/Queue Nets.

The "Needle" language developed later (Chapter 7), is based on Prolog, and has been formalised in Backus Naur Format (Appendix-F).

3.3 Execution of a Simulation

The simulator translates the model specification and executes the model as a computer program. The simulator runs interactively, displaying each event and system state (Figure 3.1). The passage of time is simulated by an integer **T** which increments in randomised jumps. Branching decisions are chosen randomly to test out all event sequences. On completion of a run, the simulator generates report statistics which analyse the performance of the model.

3.4 Underlying Concurrent Process Model

The simulator accepts application models specified as timed Petri nets. Internally the simulator converts these to a concurrent process model before execution. Each place in the Petri net is represented by a process. A different type of process is available for each simulation function. Figure 4.1 illustrates the protocol model, built from eight concurrent places. Figure 3.1 displays execution of this model, (with two extra stopwatch places).

Each process was initially given one input and one output port, but this was found to be inadequate for the modelling features provided. An extra pair of **alternate** ports was added to the standard structure of a process, and this proved sufficient for all the standard process types defined in Appendix-A. For example (Appendix-A.2) the **METER stopwatch** place requires two input ports. In chapter 7, the Needle specification language generalises the Petri net model, to allow any number of named ports.

The Petri net specification defines a network of interprocess connections. The model executes by passing messages (tokens) between processes. Message transfer is a discrete event representing the firing of a Petri net transition. Message events are illustrated in the display (Fig. 3.1) by arcs **"-->"**, **"<--"** and alternate arcs **"~~>"**, **"<~~"**. An alternate arc is simply an arc from the secondary output port, or to the secondary input port of a place. For example the alternate output **"~~>"** of **S-node** is shown, sending a message to **Channel-to-R.**

3.5 Petri Net Components

The Petri net model is composed of four components: place, token, arc and transition described below. These are conventionally drawn as circle, dot, arrow and bar.

- **Place:**

  A place is implemented as an independent process. It has internal states which change as a function of time. Inputs and outputs allow it to interact with the other connected places.

- **Token:**

  A token is emitted by the output of a place and collected by the input of a place. A basic token contains no internal data, it provides a means of synchronising the output of one place with the input of another.

- **Arc:**

  A directed arc in a Petri net diagram indicates the transfer of a token. An arc is directed from the output of place to a transition, or from a transition to the input of a place.

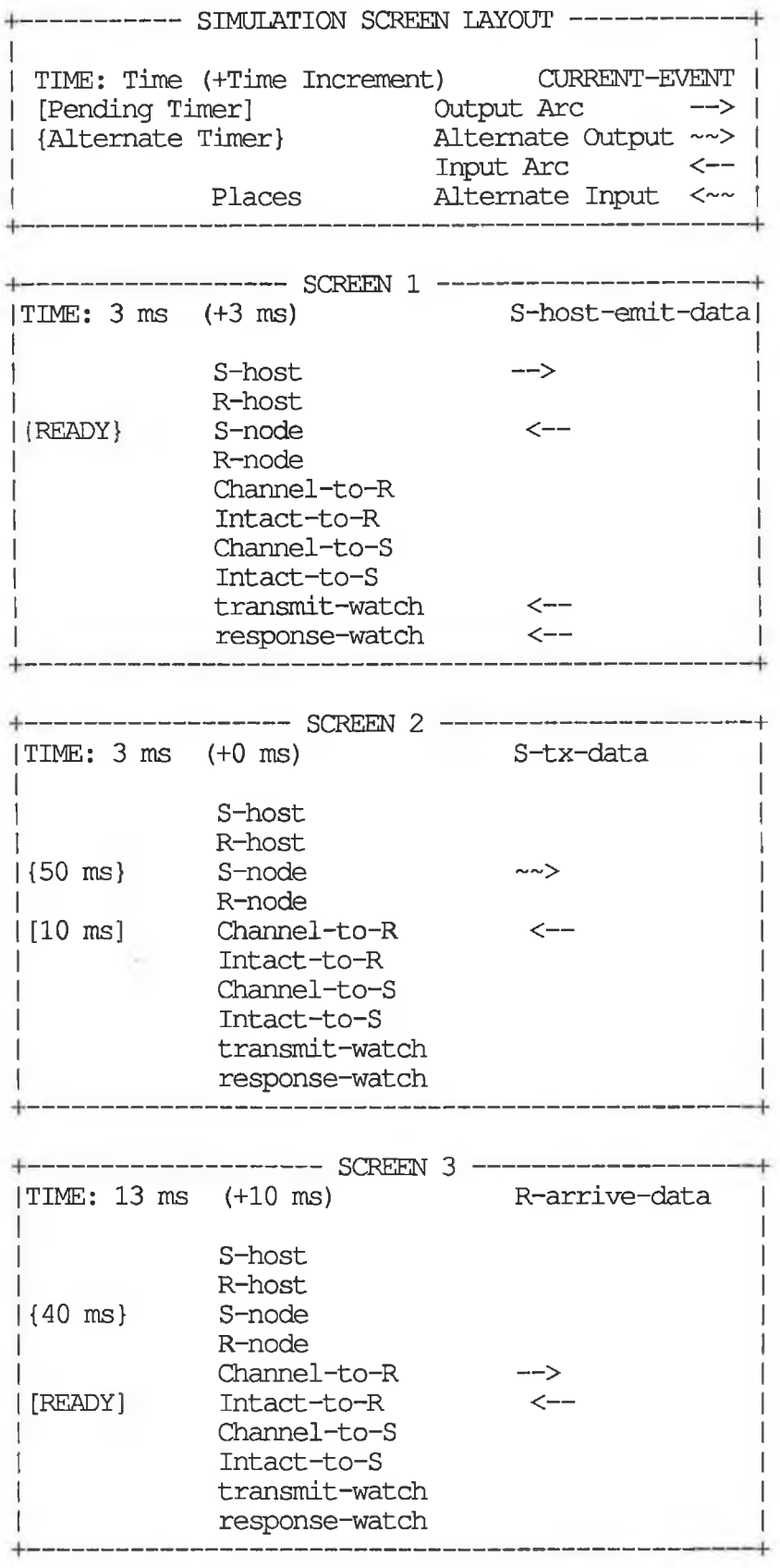
- **Transition:**

  A token transfer is instantaneous and is referred to as a transition or an event. At the instant when all its source places provide tokens an event is enabled, and passes tokens to all its destination places.

3.6 Standard Petri Net Places

A set of standard Petri net places is supplied for building timed models. They provide simulation features such as time delays and branch decisions. They are called up via the specification language, and customised by supplying parameters. A classification of standard places is given in Appendix-A, and one type, the **METER PLACE** is specified in detail. Meter places control collection of simulation statistics and provide automated generation of performance analysis reports. For instance they are used as stopwatches in the protocol simulation (Appendix-B) to measure transmission and response performances. Refer to [MCAL87] for a more complete description.

Figure 3.1 SIMULATOR STEP BY STEP EVENT SCREENS



3.7 The Simulation Environment and Report Files

The simulator processes an input text file, the specification of the application system. Execution of a model can be displayed interactively on the screen. A randomised simulation is re-run many times, and analysis reports automatically generated.

The simulator illustrates execution of model on the screen (Figure 3.1). Three successive screens are shown from the alternating bit protocol model (Figure 4.1), also listed as an event sequence report file (appendix-B.2). The event sequence is displayed step by step.

At each step the screen displays:

- o Simulation time and time increment since previous step.
- o The name of the current event.
- o Arcs from places which enabled the current event.
- o Arcs to places driven by the current event.
- o Places ready to activate future events.
- o Pending timers and their remaining time periods.

The simulator generates six different kinds of report files of which the three most useful are trace report , module report and analysis report (Appendix-B).

Trace Report: The events and time of occurance are listed in sequence. Recording is triggered by a specific event, permitting the detection of an improbable event by letting a random simulation run on the computer for a long time.

Module Report: Each place in the Petri net is reported, with appropriate statistics. For example the **ADMIT** place type simulates a randomised branching decision. It reports the number of decisions recorded, and the percentage of positive decisions.

Analysis Report: This is a table of performance measurements suitable for presentation in graph form. The simulator carries out repeated tests, automatically varying a key parameter of the application model, this is the X-axis variable. Selected measures of performance are plotted on the Y-axis (Appendix-B).

3.8 Structure of the Simulation Program

The simulator is programmed in 'C' language for a personal computer. It has a modular program structure. It executes as a sequential 'C' program, but simulates a concurrent process model. The process interface is a rigidly defined structure with two input ports and two output ports by which messages are passed. This arose from limitations in the way the program was written. An array of data structures (Appendix-C.1) maintains the internal state of each process. The simulation kernal advances the process states according to the execution rules of a timed Petri net model. Firing of transitions is represented by message passing between the concurrent processes. An unusual feature of the program is, storage of function pointers in the process data structures, to provide control of the programs execution, from the specification language.

The specification language calls up a set of standard process modules. The specification language is limited to the timed Petri net model and does not express numeric operations or modular structure. The limitations are overcome by allowing components of the application system to be programmed as 'C' modules.

Work continues in subsequent chapters to develop a more powerful model based on hierarchical numeric Petri nets. It has a specification language 'Needle' capable of describing communications protocols.

4 SIMULATION OF THE ALTERNATING BIT PROTOCOL

4.1 Communication Via an Unreliable Channel

The model simulates transmission of data packets from a sending host to a receiving host via an unreliable communications channel. A software layer represented as interface nodes takes care of transmission errors on the channel, providing a secure communications service to the hosts. The model is a simple protocol, but is intended to indicate potential applications to the performance analysis of standard protocols and layered communications software.

Figure 4.1 describes the system configuration and message paths.

S-HOST and **R-HOST** represent the two entities which require peer to peer data transfer.

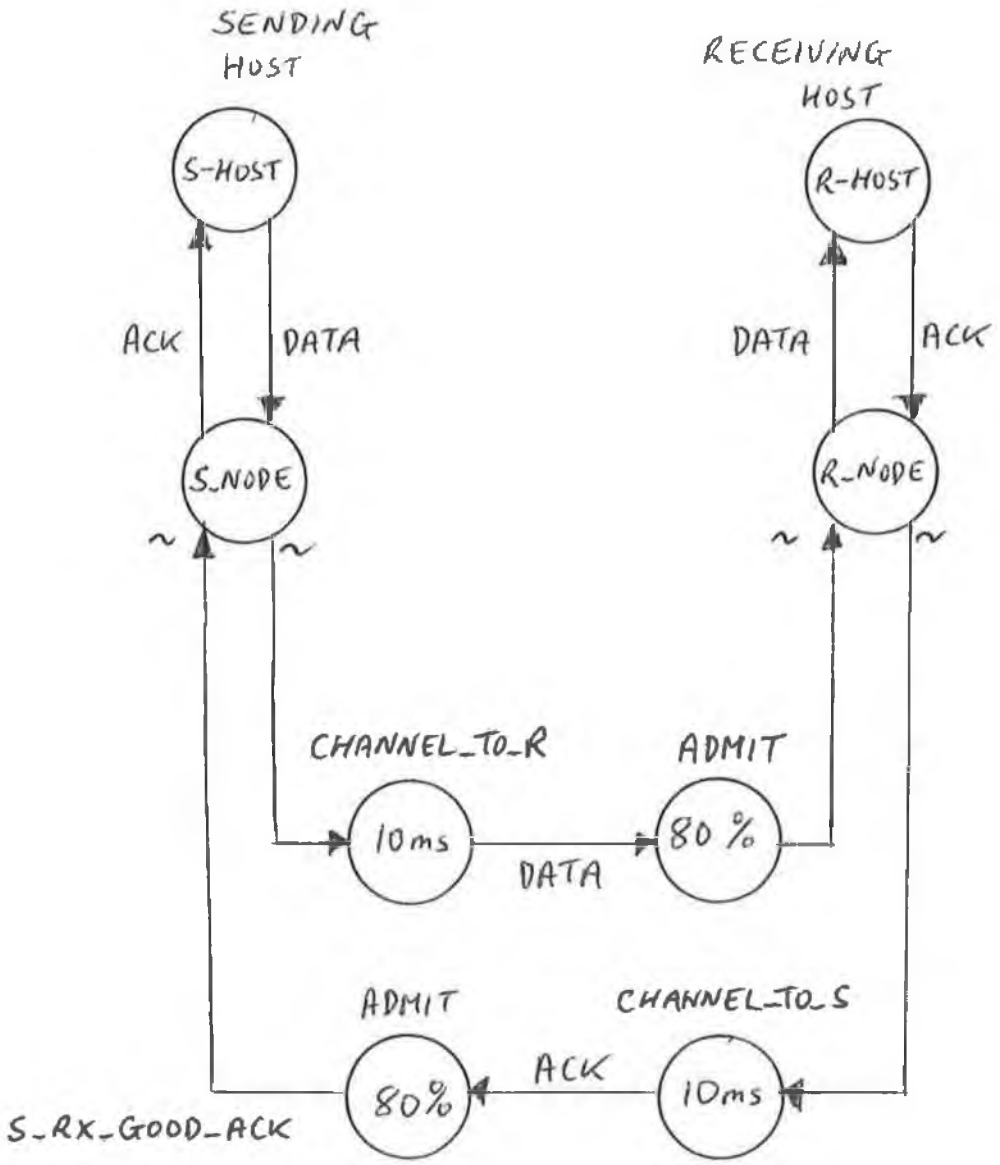
S-NODE and **R-NODE** represent the entities (software or hardware) which provide a secure interface to the channel.

CHANNEL-TO-R and **CHANNEL-TO-S** model the time delay of the communication channel in each direction.

INTACT-TO-R and **INTACT-TO-S** model the probability of uncorrupted packet transmission.

Each message event in the diagram is given a name. For example **S_RX_GOOD_ACK** indicates that the sending node (**S_**) receives (**_RX_**) an uncorrupted (**_GOOD_**) acknowledgement (**_ACK**) packet.

Figure 4.1 ALTERNATING BIT PROTOCOL MODEL



4.2 Concurrent Process Model and Implementation

The model of the alternating bit protocol is constructed from eight modules shown in the diagram (Figure 4.1). The protocol specification is listed in Appendix B.1. It is built from standard simulation components **CHANNEL, ADMIT** and **METER** (Appendix-A). In addition there are four user defined 'C' program modules:

S-HOST = Sending Host (Appendix-C.2 program listing),

R-HOST = Receiving Host,

S-NODE = Sending Node,

R-NODE = Receiving Node.

All modules have a standard interface consisting of two input ports and two output ports. The modules communicate messages via inter-port connections indicated by arrows in the diagram. The secondary ports are denoted in the diagram and in the listing by the tilde symbol **'~'**. For example the **S-NODE** and **R-NODE** modules use the primary ports to interact with the hosts and the secondary ports to interact with the channels. The messages carry data and control information between modules.

4.3 Protocol Error Recovery

The protocol transmits data packets in one direction and acknowledge packets in the other.

Between **S-NODE** and **R-NODE** the packets carry sequence numbers so that duplicate messages can be detected. The modulus-2 sequence number is a single binary digit which alternates between **0** and **1**. Hence the name **"ALTERNATING BIT PROTOCOL"**.

The protocol has two security mechanisms:

- o Timeout:

  Data packets are resent by **S-NODE** if no acknowledgement is received within a specified timeout. (In the event of **DATA** or **ACK** being lost).

- o Sequence numbers:

  The packets have a modulus 2 sequence count, so duplicate **DATA** packets are ignored by **R-NODE**. (In the event of **ACK** being lost and retransmission of the **DATA**).

The model represents a real protocol which would use checksums to detect corrupted packets. The simulation does not need to model the checksum. The **ADMIT** place introduces simulated transmission errors. The error event simulates the case of incorrect checksum calculation on the received packet, or complete failure to receive.

The packets sent from **S-HOST** carry dummy data so that **R-HOST** can verify its correct arrival. By this means we not only simulate the protocol, but verify that it is a valid protocol, delivering all the data packets in the correct sequence, and without duplication.

4.4 Protocol Performance Reports

The simulation specifies that **S-HOST** sends 50 packets to **R-HOST**. A transmission probability of 80% is specified in each direction.

The **Trace Report** (App. B.2) shows the sequence of timed events. The **Occurance Report** (App. B.3) shows how often each event occurred.

The **Module Report** (App. B.4) prints an individual summary for each module. This includes reports for two **METER** places which record timing statistics. **TRANSMIT-WATCH** measures the host to host transmit time. **RESPONSE-WATCH** measures the turnaround time for transmission and acknowledgement.

4.5 Protocol Performance Graph

Automatic generation of performance data is controlled from the model specification file (Appendix B.1). Specify the number of simulation runs at the beginning of the specification file:

```
SIMULATION
REPEAT 7
```

Modify the place definitions to specify a range of parameter values, i.e. the channel transmission probability for each simulation run:

```
PLACES
ADMIT Intact-to-R PROB 40%; 50%; 60%; 70%; 80%; 90%; 100%
ADMIT Intact-to-S PROB 40%; 50%; 60%; 70%; 80%; 90%; 100%
```

The analysis report (App. B.5) is output by the simulator. The performance graph (Figure 4.2) is generated using a spreadsheet package. The channel reliability is adjusted from 40% through to 100% to demonstrate how this affects transmission and response times.

With 100% transmission success, the host to host transmit time is 10 msec, and the transmit-response time is 20 msec. With 40% transmission success, these times increase to 92 msec and 258 msec respectively.

The performance graph demonstrates that large error rates result

in extremely long response times. It also provides for tuning of system parameters to obtain optimum performance. E.g. reducing the **S-NODE** timeout from 50 milliseconds to 25 msec improves performance at high error rates. Reducing it to less than 20 msec, the model detects collapse of the protocol.

4.6 Conclusion of Protocol Simulation

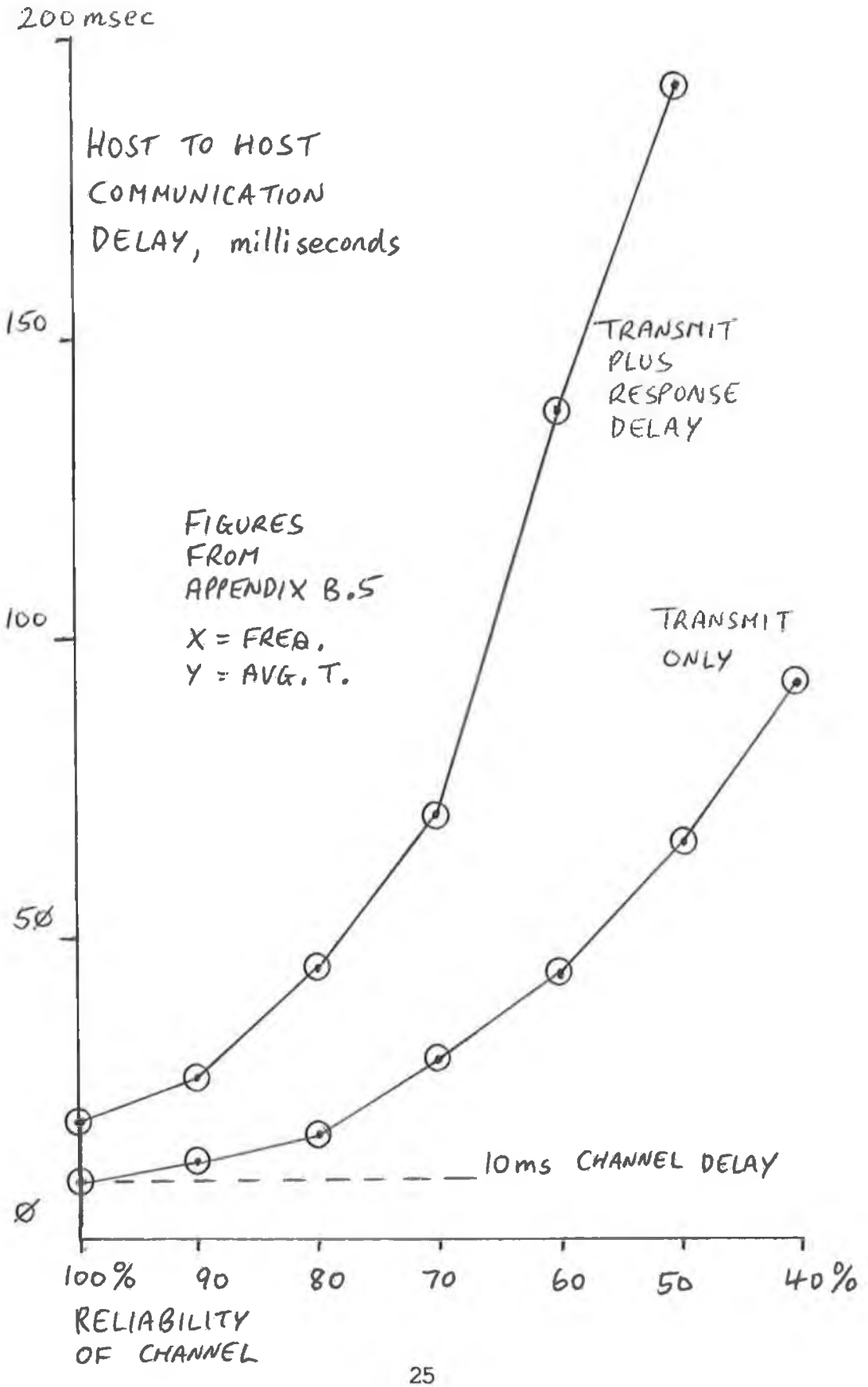
A simulation program has been developed, and the given examples show that it is capable of analysing the performance of a simple communications protocol. Previous case studies [MCAL87], show that the simulator has a broad range of application. For any chosen application, implementing a model system will have two useful outputs:

- The simulation language definition of the model contains a formal statement of the problem, including all significant time dependent relationships (Figure 4.1, Appendix-B.1).

- The single stepping event display (Figure 3.1), and the reports generated by the simulator (Appendix-B.2 to B.5) verify that the formal definition is consistent, and that the model behaves as expected.

The reports generated contain useful statistics, particularly if a real system were being modelled. The simulations are pseudo-random, so accuracy and degree of coverage of all possible event sequences will depend on the number of times that process cycles are repeated. The reports are suitable for export as ASCII files to a spreadsheet package, to produce performance graphs (Figure 4.2).

Figure 4.2 PROTOCOL PERFORMANCE GRAPH



4.7 Further work required

The simulation program began as a method of implementing timed Petri net models and evolved into a multitasking message passing simulation. It is written in 'C' and has a modular program structure which will allow it to be easily extended to incorporate new features. The inclusion and testing of 'C' program modules provides a migration path towards implementation of the model under test as working software.

Work done on the definition language was the minimum necessary to implement the chosen examples. For example any syntax errors in the specification crash the simulator. This report has not examined the state explosion problem which occurs with state-transition models of large complex systems. The problem was avoided, for example in the protocol model (Figure 4.1) where the Host and Node entities were defined internally as 'C' programs (Appendix-C.2), rather than as state transition machines.

Work is continued in subsequent chapters to implement hierarchies of entity-transition machines, where each entity may be a simple state, or a further hidden entity-transition machine. When this is acheived, effective application to complex problems may be considered, e.g. modelling of ISO OSI reference model protocols [DAGO85,FLEI87].

5 VARIOUS APPROACHES TO PROTOCOL VALIDATION

5.1 A Numerical Petri Net Validation Tool

"Protean" [BILL88] (85 references) is a protocol validation tool based on numeric Petri nets. Protocol designs may be formally specified using text or graphics. The test environment provides simulation, tracing of event sequences and reachability analysis. The advantage of the Petri net model is a solid mathematical foundation and various analysis techniques. They refer to a survey of available Petri net tools [FELD86]. For effective use, a validation tool should be integrated into a workstation environment for the computer aided design of protocols. Protean has been applied to several protocols including an OSI Transport Protocol. Since validation techniques apply to distributed systems in general, Protean is also being used to design a protocol engineering workstation.

5.2 Database Methods Overcome State Explosion Limit

The problem with complex protocol models is the huge number of possible system states, exceeding the available computational power. This is often refered to as the state-space explosion.

Protocol validation tools are being developed using database methods. The relational data model introduced by Codd in 1970 includes a relational algebra and relational calculus.

[LEE_88] have implemented a protocol verification tool on the INGRES relational database. A protocol is formally defined by its state transitions, and expressed as a data table. Database theory is

used to derive the global properties of the protocol model and detect logical errors. Since databases are designed to process large amounts of data, the technique can be applied to complex protocols.

[FRIE89] design a binary tree protocol specifically to demonstrate the state space explosion. They employ relational algebra to verify the protocol. A model with 7540 reachable global states was verified in 15 hours on a microcomputer. They simulate employing the power of a 100 processor "hypercube multicomputer" to search the state space in a fraction of a second. For application to real protocols, the "Finite State Machine" model requires extension and the search algorithm should use backtracking to locate undesirable states.

5.3 Logic Programming as a Validation Environment

Some researchers are turning to logic programming systems as an environment for validation of formal specifications. A profitable research feedback situation arises because logic programming itself becomes the object of study by formal methods [MURA88,PETE89]. This will result in new models of logic programming systems, including performance models. Inefficient use of computer hardware is a major obstacle to the commercial exploitation of logic programming. This cycle of research will lead to the development of viable logic programming systems running on specially designed computer architectures. The Prolog language [CLOC87] has most potential, and is used for the development of fifth generation computers [MOTO85].

5.4 Using Temporal Logic with Concurrent Prolog

There are alternatives to the Petri net approach. [SANT88] use "Temporal Logic" as a specification language with "Concurrent Prolog" for implementing prototypes. This approach provides an unrivalled method of protocol specification. Their paper lists a full specification and implementation of the alternating bit protocol. They are successful in specifying the behavior of the protocol, including timer initiated retransmission of lost messages.

5.5 Validation of a Modular Specification

[REED88] illustrates a hierarchical system of concurrent modules which communicate according to "Communicating Sequential Process" semantics [HOAR78]. Each module has a set of ports via which it links to its neighbours. A proof system verifies the safety and liveness properties of a specification. Their approach allows verification of an individual module without concern for the internal structure of its neighbours.

5.6 Development of the "TRAV" Validation Tool

A Petri net specification language and validation tool were developed in Prolog. The specification language "Needle" specifies Petri nets which have a modular structure and carry out numeric operations. A specification of the alternating bit protocol is validated by reachability analysis, (an exhaustive search of all

possible event sequences). The "TRAV" tool is written in Turbo Prolog for a personal computer, and makes extensive use of Prolog's backtracking, list processing and database facilities.

[KIM_87] (among many others) also use numeric Petri nets for protocol validation.

5.7 Conclusions for Protocol Validation

For ease of analysis, protocol models must be based on a sound mathematical model. The Petri net is such a model, and with various extensions is suitable for modelling real protocols. The validation tool should be an integrated part of a protocol development environment. The environment should control modular construction of complex specifications from reuseable modules. Special computer architectures may be required to carry out efficient analysis of complex models.

Several researchers are implementing logical formalisms, such as Petri net theory or temporal logic, using the Prolog programming environment. This could indicate a general trend in the development of computer aided tools for various branches of scientific research. A specialised logical formalism (a scientific model) is implemented in the Prolog environment, creating a powerful tool for the validation of theories and models.

6 SCHEMATIC DESCRIPTION TECHNIQUE

6.1 SDL Model of Alternating Bit Protocol

This chapter examines a protocol model from [CAVA87] specified in the CCITT's "Specification Description Language", **"SDL"**. The language describes state transition machines. States are identified by the keyword **STATE**, and transitions by the keyword **NEXTSTATE**. An SDL simulation tool verifies the protocol, which is designed to recover from communication errors.

The SDL specification is listed in Appendix-D. It models transmission of data from a **SENDER** process to a **RECEIVER** process via a **CHANNEL** process. Data flow errors are introduced on the channel by "yes/no" input from the user.

A summary of the component terms gives a quick idea of how the protocol model is represented:

sender process	-	transmits data, receives acknowledge
receiver process	-	received data, tranmits acknowledge
channel process	-	communicates between sender and receiver
idle state	-	do nothing until there is data to transmit
wait state	-	wait to receive a message from the channel
0,1	-	modulus 2 sequence counter (ALTERNATING BIT)
dm	-	data message
am	-	acknowledge message
content	-	a message passing through the channel

6.2 Extensions to Petri Net Model

The eventual goal is to implement the protocol specification in Prolog instead of SDL. The first step is to transform the SDL model into a precise graphical representation, based on extensions to the Petri net model. The extensions are:

- A modular Petri net, using top down design to specify the system as a hierarchy of modules.

- A numerical extension to the Petri net, where each token (or message) carries an integer value.

6.3 Schematic Protocol Representation

6.3.1 Hierarchical System Model

Figure 6.1 shows the hierarchical model of the protocol system. The main module contains four sub modules: **SENDER**, **CHANNEL_DATA**, **CHANNEL_ACK**, and **RECEIVER**. Only two levels are required, but this structure is applicable to complex multilayer systems.

Appendix-E.2 gives a schematic of the main module, defining the overall structure of the communicating system. Instead of the SDL bidirectional channel, I have used separate channels, **CH_DATA** and **CH_ACK**, for the message flow in each direction. Four transitions define the direction of message flow within the system:

s_tx - sender transmits data

r_rx - receiver receives data

r_tx - receiver transmits acknowledge

s_rx - sender receives acknowledge

The complete graphical specification of the system is defined by four schematics (Appendix-E.2 to E.5). The **MAIN** module and **CHANNEL** module declare the system and its circumstances, i.e. communication over unreliable channels. The **SENDER** module and **RECEIVER** module declare the protocol entities which implement the error correcting algorithm.

6.3.2 Channel Module

The channel module (Appendix-E.3). It's external features are simply its identity **C-MOD**, and two interface ports **IN** and **OUT**. Internally the module contains three transitions:

tx_msg - transfer a message from **IN** to **OUT**

tx_err - take an input message, but force an error in its contents before transferring it to output.

lose_msg - accept a message from the input and lose it.

The three transitions are mutually exclusive, and of equal priority. When a message arrives at the input port of the channel, any transition may occur, giving one of the three possible results. The **MAIN** module contains two instances **CH_DATA** and **CH_ACK** of the channel module. The SDL channel (Appendix-D.6) contained 5 states and 12 transitions. The channel module **C-MOD** is much simpler, containing only 3 transitions and no states.

6.3.3 Sender Module

The sender module **S-MOD** (Appendix-E.4), corresponds to the SDL sender process. The schematic represents a Petri net with four places: **IDLE0, WAIT0, IDLE1, WAIT1;** corresponding to the four SDL states. At any instant, the module contains one token. The place containing the token indicates the state of the protocol entity. The initial state is **IDIE0** - do nothing until there is data to transmit. Six transitions define how the token moves about the module. The diagram shows multiple arcs to the boundary of the module. This is simply for clarity, there are in fact only two ports, labelled **IN** and **OUT.**

6.3.4 Receiver Module

The receiver module **R-MOD** (Appendix-E.5) corresponds to the SDL receiver process. The schematic represents a Petri net with two places: **WAIT0** and **WAIT1;** corresponding to the two SDL states. At any instant, the module contains one token. The place containing the token indicates the state of the protocol entity. The initial state is **WAIT0** - wait to receive a message from the channel. The module has six internal transitions, and two ports.

6.4 Numerical Petri Net Model

The diagrams use a numerical Petri net extension. This is indicated by integers **[0]**, **[1]** and **[-1]** associated with each transition. The integers are represented in two senses:

- On an **INPUT ARC** to the transition. The transition is only enabled if the incoming token matches the specified value. A transition with multiple input arcs will only be enabled when all arcs provide acceptable tokens.

- On an **OUTPUT ARC** of the transition, the specified value will be inserted into the transmitted token. When no value is specified on the arc, the default value is a merge function of the input values.

When a transition with multiple incoming arcs occurs, a merge function is performed on the token values. The function is:

maximum_integer(token_1,..., token_N).

The result is the default value for tokens on outgoing arcs.

As an example consider the transition **S0_RX_A1** in the sender module (Appendix-E.4) The name indicates that the sender is in the **WAIT0** state, and receives an acknowledge message with sequence count **'1'**. The event will occur if the **WAIT0** place contains a token, and if a token of value **[1]** arrives on the input port. The result of the transition is that a token will be returned to the **WAIT0** place, and another token of value **[0]** will be sent to the output port of the module.

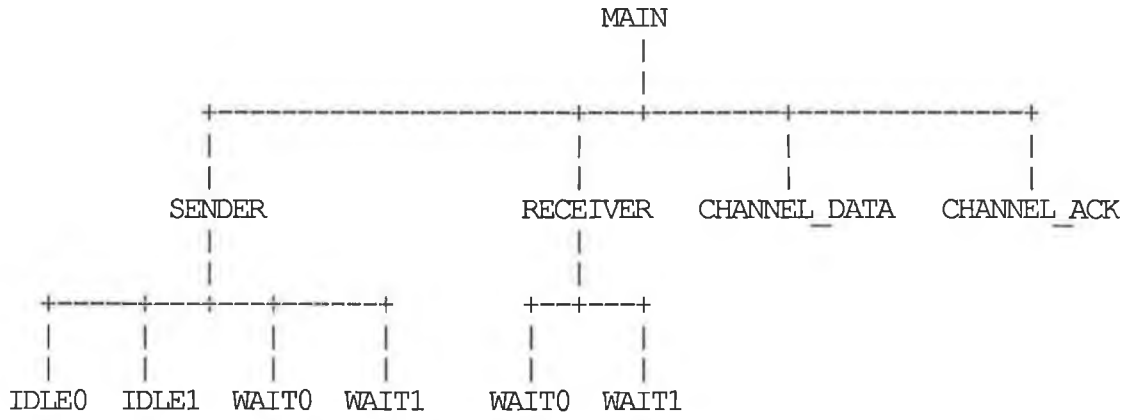
6.5 Summary of Schematic Representation

That completes schematic specification of the protocol system. The schematics use a hierarchical numerical Petri net model. It is not immediately clear how the protocol will function. Operation may be analysed mentally, by tracing through the arcs of the schematics, noting the movement of tokens. It is the function of this project to automate the analysis process. The method of automation, is to construct an executable logical model from the schematics.

FIGURE 6.1

HIERARCHICAL STRUCTURE OF THE PROTOCOL

The model system is designed as a hierarchy of modules. **MAIN** is the top level (or system, or root module). The **SENDER** and **RECEIVER** modules, along with the **CHANNEL** modules are at the next level. Elementary Petri net places are at the lowest level. The transitions are also in the hiererarchy, though not represented here.



7 THE "NEEDLE" SPECIFICATION LANGUAGE

7.1 Specification and Validation Using Prolog

I have shown in the previous chapter, how a protocol model may be represented by diagrams of state transition networks. This chapter describes the transformation of these diagrams into a logical specification. The specification is written using an appropriate logic programming language - "PROLOG" [CLOC87]. In order to do this, I have designed a block structured specification language "NEEDLE" (Appendix-F), and show how the specification may be written as a Prolog program. The language name "Needle" is chosen, on the analogy that a network of places is "stitched" together by the interconnecting arcs.

The underlying mathematical model is a numerical hierarchical Petri net [BILL88]. I have written a reachability analysis program "TRAV.PRO", which validates the specification, by TRAVersing all possible execution paths. The program makes extensive use of Prolog's list processing, backtracking, and database facilities.

7.2 Petri Net Analysis of Needle Specification

The analysis tool TRAV.PRO translates the Needle specification and carries out reachability analysis. Validation experiments are controlled interactively by commands, or by menus:

Main analysis commands	-	**spec, search**
Terminal conditions	-	initial, end_option, end_state
Event control	-	avoid, occur
Search control	-	depth, permit_loops, first_result
Display control	-	show, tree, track

Analysis commands:

The **SPEC** command translates the modular specification into a numerical Petri net model.

The **SEARCH** command executes event sequences, and carries out reachability analysis.

The menu interface is described in appendix-H. The "TRAV" program also has a manual describing entry of the same commands in dialog mode.

7.3 Structure of Needle Specification in Prolog

A specification file e.g. (Appendix-G.1) has two sections:

- o The predicates section, only a few lines.
- o The clauses section, containing the specification.

A Prolog clause has the general structure:

GOAL :- SUBGOAL, SUBGOAL, ... SUBGOAL.

A model is specified by a sequence of subgoals. Every entity in the model (module, place or transition) requires a subgoal to declare it. The specification takes the form:

spec :- begin_spec, SUBGOAL, ..,SUBGOAL, end_spec.

You may ask "But doesn't Prolog implement backtracking, so how can the subgoals be described as a sequence ?". In answer, all subgoals of the **SPEC** clause are designed to return "true", backtracking would only occur if they returned "false". There is enough backtracking done elsewhere in the program to keep the Prolog interpreter happy!

Execution of the clause stores the specified model in the Prolog database. The first subgoal **BEGIN_SPEC** prepares the database. Each subgoal executes in turn, storing the declaration of an entity. The final subgoal **END_SPEC** processes the contents of the database, verifying the specification for "Needle" syntax errors.

7.4 Needle Programming Statements

The structure of a model system is built of five static entities:

module,	-	Needle program block
ports,	-	Inputs and outputs of a module
place,	-	Instance of module, as a concurrent process
transition,	-	Transfers tokens between places
arc.	-	Connection between place and transition

And one dynamic entity:

token.	-	Message carrying an integer value.

There are nine program statements used to declare the static entities in a Needle program:

(a) module(module_name)

(b) end(module_name)

(c) port(port_name)

(d) place(module_type,place_name)

(e) transition(transition_name)

(f) from(input_arc)

(g) get(input_arc,value)

(h) to(output_arc)

(i) put(output_arc,value)

Specification of a module may use all statements. The specification is block structured, each module is a program block.

7.5 Modular Program Block

A module is described by a program block. The subgoal **module()** begins the block, and the subgoal **end()** ends the block. Places, transitions and port names are local to the module in which they are declared. The general structure of a module is:

```
module(module_name),
    ports...
    places...
    transitions and arcs...
end(module_name),
```

E.g. the **MAIN** module of the alternating bit protocol (Appendix G.1) is declared:

```
module(main),
  place(s_mod,sender),
  place(r_mod,receiver),
  place(c_mod,ch_data),
  place(c_mod,ch_ack),
    transition(s_tx), from(sender),    to(ch_data),
    transition(r_rx), from(ch_data),   to(receiver),
    transition(r_tx), from(receiver), to(ch_ack),
    transition(s_rx), from(ch_ack),    to(sender),
end(main),
```

The main module has no ports, because it is the highest level module, and has no peers to communicate with.

7.6 Declaring a Petri Net Place

General declaration of a place:

place(module_type,place_name)

Example: place(c_mod,ch_data)

This subgoal declares a Petri net place. The place is local to the module in which it is declared.

argument 1 - the module which defines the internal structure, **"elementary"** specifies no internal structure.

argument 2 - assigns a name to the place.

7.7 Declaration of Transition and its Arcs

Five types of subgoal are used in declaring a transition:

1. transition(transition_name),
2. from(input_arc),
3. get(input_arc,value),
4. to(output_arc),
5. put(output_arc,value)

For example the **SEND_D0** transition in Appendix-G.1, declared:

transition(send_d0), get(idle,0),
put(wait,0), put(out,0),

It inputs one token of value '0', and emits two tokens of value '0'. It is local to the sender module in which it is declared.

An input arc defines the source of a token:

from(source) - source may be a place, local to the module, or may be a port of the module.

get(source,value) - The second argument specifies an integer value which the available token must contain in order to enable the transition.

An output arc defines the destination of a token:

to(dest) - destination may be a place, local to the module or may be a port of the module.

put(dest,value) - The second argument specifies an integer value to be carried in the output token.

7.8 Conclusion - Needle Replaces the SDL Language

In conclusion, it is shown (Chapter 6) that an SDL specification (Appendix-D) of the alternating bit protocol, can be transformed manually into a graphical Petri net representation (Appendix-E). Secondly (this chapter), the Petri net is easily defined using the Needle specification language in a Prolog environment. Thus an SDL protocol specification can be translated to a Needle language specification (appendix-G). Needle is a Petri net language, giving potential access to the analytic tools of Petri net theory. Reachability analysis is the specific Petri net tool, which we will apply in the next chapter.

8 "NEEDLE" SPECIFICATION OF ALTERNATING BIT PROTOCOL

8.1 Sender and Receiver Modules

The alternating bit protocol model is specified in the Needle language. Listings 8.1 and 8.2 specify two modules, **SENDER** and **RECEIVER**. They are an exact translation of the schematics (Appendix E.4,E.5). The modules are state transition machines which together implement the error correcting algorithm of the protocol.

The **SENDER** module has four places corresponding to its four possible states.

idle0 - Doing nothing, sequence counter is 0

wait0 - Pending ack message, sequence counter is 0

idle1 - Doing nothing, sequence counter is 1

wait1 - Pending ack message, sequence counter is 1

The **RECEIVER** module has two places corresponding its two possible states.

wait0 - Pending data message, sequence counter is 0

wait1 - Pending data message, sequence counter is 1

The modules should be compared with their original SDL specification (Appendix-D).

The Needle to SDL translation of the main terms is:

module() = PROCESS

place() = STATE

transition() = NEXTSTATE

8.2 Integer Tokens Reduce Number of Places

I have made use of the properties of numerical Petri nets to compress the representation, (Appendix-G.1). The token within a place can contain an integer value. Values **0** and **1** are used to represent the sequence counter, reducing the number of places required. The **sender** module has only two places: **idle** and **wait**, but effects four distinct states. The **receiver** module has only one place: **wait**, but effects two distinct states.

Examine one transition, as an illustration of how the model functions. Take the **send_d0** transition, in the sender module of Appendix-G.1:

```
transition(send_d0), get(idle,0),
                     put(wait,0), put(out,0),
```

It has one input arc **get(idle,0)**, i.e. is only enabled if the **idle** place provides a token with value **0**. On activition, tokens are put on the two output arcs. A token with value **0** is put to the **wait** place, and a token with value **0** is put to the **out** port of the module. A subsequent transition **s_tx** in the **main** module is thus enabled. It accepts the token output by the **sender** module, and in isolation of the internal behavior of **sender**. That is the advantage of abstraction, the sender module is abstracted in the next level, as a black box with simply an input port and an output port. The **s_tx** transition forwards the token to the **ch_data** module, and so on.

8.3 Hierarchical and Flat Petri Net models

The modular specification is a hierarchical Petri net model (Appendix-G.2). It is structured as a tree of modules, and has a multi-level network of interconnecting transitions.

The **spec** command translates the specification into a single level (flat) Petri net model. The flat Petri net is stored in the Prolog database as a one dimensional list of places, and a one dimensional list of transitions (Appendix-G.3).

8.4 Translation to Flat Petri Net Model

Transformation from the hierarchical model to the flat model maintains the identities of places. Each place in the place list (Appendix-G.3) is identified by its path.

The path of a place is the list of modules by which it is descended from the main module. For example **[]** is the path of the top level main module. **[sender,idle]** is the path of the **idle** place in the **sender** module. The list manipulation features of Prolog are ideal for representing paths as lists **[...]** of symbols.

Similarly a specified transition is identified by the path of the module which contains it, and the name of the transition. For example **[ch_data]tx_msg** and **[ch_ack]tx_msg** are two distinct transitions, because they are in different instances of the channel module.

8.5 Efficiency of Search Algorithm

The flat Petri net model is an intermediate stage in analysis of the specification. The **search** command traverses the flat model. It is not essential to flatten the specification before analysis. If the search algorithm were being implemented on a parallel computer, it would be more efficient to process the hierarchical model directly. because the different protocol entities would execute concurrently.

I found the flat search to be much faster than the hierarchal, presumably because I am carrying out the testing on a single processor computer.

8.6 Analysis of the Protocol Event Sequence

An analysis of the protocol is carried out to investigate its properties under conditions of error free communications. A search of event sequences is carried out interactively. Appendix-G.4 shows the resultant display. The display has three sections:

Test conditions:
The test conditions detail the environment as set up by previous test control commands. The model initially has two tokens of value **0**. A final state is required, with **[sender,idle]** place containg a **0** token. Error events **lose_mg** and **tx_err** are to be avoided on the search path, thus investigating the protocol under error free conditions.

Test result:

Indicates that the final condition is reached after 18 executed events. Shows that the final state contains the same pattern of tokens as the initial state. The protocol model has executed a complete cycle, completing the transfer of two data packets.

Valid event sequence:

Events 1 to 9 show communication with sequence count of **0**.

Events 10 to 18 show communication with sequence count of **1**.

If we are interested in traffic on the channels, then four events are significant:

```
3 [ch_data] tx_msg      /* SENDER --> DATA 0 --> RECEIVER
7 [ch_ack] tx_msg       /* SENDER <-- ACK 0  <-- RECEIVER
12 [ch_data] tx_msg     /* SENDER --> DATA 1 --> RECEIVER
16 [ch_ack] tx_msg      /* SENDER <-- ACK 1  <-- RECEIVER
```

They show: **DATA** transmission from sender to receiver, and **ACK** transmission from receiver to sender.

8.7 Conclusion - The Needle Specification is Executable

In conclusion, the analysis tool TRAV.PRO has succeeded in translating a Needle specification and executing it as a sequence of events (Appendix-G.4). The event sequence shows that under error free communications, the protocol transfers two data messages from sender to receiver, and returns both protocol entities to their initial state. Further aspects of the protocol behaviour can be investigated in a similar manner.

Listing: 8.1 SENDER MODULE SPECIFIED IN NEEDLE

This is an exact specification of the schematic (Appendix-E.4).

The sender module is defined with places:

idle0 - Doing nothing, sequence counter is 0
wait0 - Pending ack message, sequence counter is 0
idle1 - Doing nothing, sequence counter is 1
wait1 - Pending ack message, sequence counter is 1

Presence of a token in a place indicates the state

and sequence count of the module.

```
/* Sender module */

module(s_mod),

  port(in), port(out),

    place(elementary,idle0),

    place(elementary,idle1),

    place(elementary,wait0),

    place(elementary,wait1),

      transition(send_d0), from(idle0),
                           to(wait0), put(out,0),

      transition(send_d1), from(idle1),
                           to(wait1), put(out,1),

      transition(s0_rx_err), from(wait0), get(in,-1),
                             to(wait0), put(out,0),

      transition(s1_rx_err), from(wait1), get(in,-1),
                             to(wait1), put(out,1),

      transition(s0_rx_a1), from(wait0), get(in,1),
                            to(wait0), put(out,0),

      transition(s1_rx_a0), from(wait1), get(in,0),
                            to(wait1), put(out,1),

      transition(s0_rx_a0), from(wait0), get(in,0),
                            to(idle1),

      transition(s1_rx_a1), from(wait1), get(in,1),
                            to(idle0),

 end(s_mod),
```

Listing: 8.2 RECEIVER MODULE SPECIFIED IN NEEDLE

This is an exact specification of the schematic (Appendix-E.5).

The receiver module is defined with places:

wait0 - Pending data message, sequence counter is 0

wait1 - Pending data message, sequence counter is 1

Presence of a token in a place indicates the state and sequence count of the module.

```
/* Receiver module */
module(r_mod),
  port(in), port(out),
    place(elementary,wait0),
    place(elementary,wait1),
      transition(r0_rx_err), from(wait0), get(in,-1),
                             to(wait0), put(out,1),
      transition(r1_rx_err), from(wait1), get(in,-1),
                             to(wait1), put(out,0),
      transition(r0_rx_d1), from(wait0), get(in,1),
                            to(wait0), put(out,1),
      transition(r1_rx_d0), from(wait1), get(in,0),
                            to(wait1), put(out,0),
      transition(r0_rx_d0), from(wait0), get(in,0),
                            to(wait1), put(out,0),
      transition(r1_rx_d1), from(wait1), get(in,1),
                            to(wait0), put(out,1),
  end(r_mod),
```

9 ANALYSIS OF PROTOCOL ERROR RECOVERY

9.1 Event Sequence Recovers from Data Error

The protocol model was formally specified using Needle in Appendix-G.1. This chapter describes the results of analysis carried out on the protocol model. The protocol specification ALT.PRO and the analysis tool TRAV.PRO are both written in Prolog.

Appendix-I.1 taken from the screen display, shows the results of interactive analysis by the **search** command. The search algorithm discovers the displayed event sequence, by rigorous search of all all events reachable from the initial state. The search is a depth first algorithm which makes use of Prolog's in-built backtracking.

Test conditions force a transmission error on the data channel. The results show that the protocol recovers from the error, and reaches the required final state after 17 events. The event sequence which achieved error recovery is listed.

Event **1, [sender]send_d0,** transmission of a data message with sequence number [0].

Event **3 [ch_data]tx_err** is the forced error, a transmit error on the data channel, which puts a [-1] in the message.

Event **5** shows detection of the error by the protocol machine of the receiver entity.

Event **9,** the sender gets an acknowledge message with unexpected sequence number [1] , so retransmits the original data message.

Event **13,** the receiver acknowledges receipt of a correct message.

Event **17,** the sender gets an acknowledge message with the correct sequence number [0].

Appendix-G.4 showed the same state being reached in only 9 events, if no transmission errors occured.

Conclusion: The protocol recovers from a data transmission error but at the expense of doubling the amount of traffic and processing required for that message. This conclusion is for one message, it doesn't extrapolate to a performance prediction for large message volumes.

9.2 Transmission Errors in Both Directions

Appendix-I.2, This is a repetition of the previous test, which forced a data error **[ch_data]tx_err.** This test also forces the occurance of **[ch_ack]tx_err.** The **show** command (Appendix-H.6) has been used here, to zoom in on the transmission channels. Only events in the modules **[ch_data]** and **[ch_ack]** are selected for display. The listing shows two experiments; the first forcing an error on the data channel, and the second forcing an error on the acknowledge channel.

Conclusion: The protocol recovers when an error occurs either in a data message or in an acknowledge message.

9.3 Reachability Tree of The Protocol

Figure 9.1, the test conditions specify a complete search of all events reachable from the initial state. The **end_option(cycle)** terminates each sequence when it loops back to a previous state. The option **tree(yes)** presents the results as a reachability tree. Only transmission events are shown, those in the **[ch_data]** or **[ch_ack]** modules.

Figure 9.1 Reachability Tree of Cyclic Sequences

```
depth(30), search depth          /* Test conditions */
Initial state:
  ["sender","idle"] 0
  ["receiver","wait"] 0
end_option(cycle)
Show nodes:
  ["ch_data"]
  ["ch_ack"]
Option: tree(yes), display reachability tree

--+-tx_msg--+-tx_msg--+-tx_msg--+-tx_msg
  |         |         |         |
  |         |         |         +-err_msg-+-tx_msg
  |         |         |                   |
  |         |         |                   +-err_msg
  |         |         |
  |         |         +-err_msg-+-tx_msg
  |         |                   |
  |         |                   +-err_msg
  |         |
  |         +-err_msg-+-tx_msg
  |                   |
  |                   +-err_msg
  |
  +-err_msg-+-tx_msg
            |
            +-err_msg

Search for all solutions complete -
9 valid sequences end in cycles.
Depth sufficent to find all cycles.
```

The algorithm reports that the search depth was sufficient, therefore the tree is complete. The tree should be read as an execution sequence, from left-hand root node to the right-hand leaf nodes. Nine valid execution paths can be seen. The first path represents a conversation of four **tx_msg** messages on error free channels. The last path represents **err_msg** on the data and on the acknowledge channels.

9.4 Lost Messages Cause Deadlock

The protocol is explored for all event sequences which lead to deadlock. A search of depth 30 found eight deadlock conditions. Figure 9.2 displays the results as a reachability tree. The leaves of the tree (right hand side) indicate the last transmission event before deadlock. It can be seen that in all cases **lose_mg** deadlocks the model, and no other event leads to deadlock. It can be concluded that the alternating bit protocol, as specified in Appendix-G.1, is unable to recover from loss of messages during transmission.

Figure 9.2 Reachability Tree of Deadlock Sequences

```
depth(30), search depth          /* Test conditions */
Initial state:
  ["sender","idle"] 0
  ["receiver","wait"] 0
end_option(deadlock)
Show nodes:
  ["ch_data"]
  ["ch_ack"]
Option: tree(yes), display reachability tree.
Option: permit_loops(no), search until loop.

--+-tx_msg--+-tx_msg--+-tx_msg--+-err_msg---lose_mg
  |         |         |         |
  |         |         |         +-lose_mg
  |         |         |
  |         |         +-err_msg---lose_mg
  |         |         |
  |         |         +-lose_mg
  |         |
  |         +-err_msg---lose_mg
  |         |
  |         +-lose_mg
  |
  +-err_msg---lose_mg
  |
  +-lose_mg

Search for all solutions complete -
8 valid sequences to deadlock.
Depth sufficent to find all cycles.
```

10 PROTOCOL MODEL WITH TIMEOUT FACILITY

10.1 Specification of Timeout Transitions

The protocol model was formally specified in Appendix-G.1. Chapter-9 showed that the protocol doesn't work if messages are lost in transmission. To fix this, add a timer module **t_mod** to the specification (Appendix-J.1). The sender module **s_mod** has two new transitions **s0_timeout** and **s1_timeout**, activated by the timer module, to retransmit lost messages.

The underlying Petri net is still a purely logical model. It has NO extensions to simulate the passage of time. The timer module is a quick fix to correct the protocol without overhaul of the underlying model. The logical model has only two measurements of time:

- no time at all
- a long time indeed

and these are sufficient to solve the timeout problem.

A new place called **deadlock** is embedded in the model environment. It is a public place, any transition can declare an arc from it. During analysis, if the search algorithm detects a deadlock situation, it puts a token in the deadlock place. The token represents that **"a long time indeed"** has passed. The token activates any connected transitions, releasing the model from deadlock. In the protocol model, the timer module **t_mod** accepts the deadlock token, and uses it to initiate timeout transitions.

10.2 Timer Retransmits Lost Data Message

Appendix-J.2 shows test conditions forcing loss of a **data** message. The results show that the protocol recovers and reaches the final state after 13 events. The event sequence is listed:

Event 1, **[sender]send_d0,** transmission of a data message with sequence number [0].

Event 3 **[ch_data]lose_mg** is the forced event.

Event 4 shows elapse of the timer in the sender module.

Event 5 is the timeout event which retransmits the data msg.

Event 9, the receiver acknowledges receipt of a correct message.

Event 13, the sender gets an acknowledge message with the correct sequence number [0].

10.3 Timer Retransmits Lost Acknowledge Message

Similar to the previous test, but tests the case of a lost **ack** message. The test conditions force the occurance of **[ch_ack]lose_mg**. The **show** command (Appendix-H.6) has been used here, to display events only in the channel modules, and in the module **[sender,timer]**.

```
Initial state:                          /* Test conditions */
  ["sender","idle"] 0
  ["receiver","wait"] 0
Final conditions:
  ["sender","idle"] 1
Show nodes:                                 /* Show events in
  ["ch_data"]                                  channel and timer
  ["ch_ack"]                                   nodes only     */
  ["sender","timer"]
Occur event sequence:
  1 ["ch_ack"] lose_mg                      /* Force a lost ACK

17 events to final state:               /* Test result */
  ["receiver","wait"] 1
  ["sender","idle"] 1

Valid event sequence:
  3 ["ch_data"] tx_msg                    /*   DATA 0 --> RECEIVER
  7 ["ch_ack"] lose_mg                    /*   LOST <-- ACK 0
  8 ["sender","timer"] elapse             /*   SENDER TIMEOUT OCCURS
  11 ["ch_data"] tx_msg                   /*   DATA 0 --> RECEIVER
  15 ["ch_ack"] tx_msg                    /*   SENDER <-- ACK 0
```

Conclusion:

We have successfully modified the protocols error recovery algorithm by the addition of two timeout transitions to the sender module. This provides recovery from either a lost **data** message, or a lost **ack** message, situations which previously caused deadlock.

10.4 Prove Absence of Protocol Deadlock

Appendix-J.3. The protocol is explored by reachability analysis. specifying a search for all event sequences which lead to deadlock. With a search depth of 30 events, no deadlock situations are found. The algorithm reports that the search depth was sufficient to complete the search. Since the backtracking search algorithm completes an exhaustive search of all event sequences, we can conclude that no deadlock states are reachable from the given initial state.

Conclusion -

Beginning from the given initial state, the protocol model is deadlock free. Thus we have defined and proven a protocol which recovers from both messages with errors, and lost messages.

10.5 Reachability Tree of Protocol with Timeout

The test conditions (Figure 10.1) specify a search of all events reachable from the initial state. Each explored sequence is terminated when it loops back to a previous state of the model. This test was previously done in Figure 9.1, but now we have a corrected protocol machine capable of recovery from lost messages. The test reports that the reachability tree is complete.

The analysis reports 21 distinct event sequences, all of them taking the protocol machine to either a successful, or a recoverable state. Every joint on the tree corresponds to one state of the model system. In the interest of simplicity the tree only shows events on the communications channels, and omits the channel names.

Figure 10.1 Reachability Tree of Cyclic Sequences

```
depth(30), search depth      /* Test Conditions
Initial state:
  ["sender","idle"] 0
  ["receiver","wait"] 0
end_option(cycle)
Show nodes:
  ["ch_data"]
  ["ch_ack"]
Option: tree(yes), display reachability tree.

--+-tx_msg--+-tx_msg--+-tx_msg--+-tx_msg
  |         |         |         |
  |         |         |         +-err_msg-+-tx_msg
  |         |         |         |         |
  |         |         |         |         +-err_msg
  |         |         |         |         |
  |         |         |         |         +-lose_mg
  |         |         |         |
  |         |         |         +-lose_mg-+-tx_msg
  |         |         |                   |
  |         |         |                   +-err_msg
  |         |         |                   |
  |         |         |                   +-lose_mg
  |         |         |
  |         |         +-err_msg-+-tx_msg
  |         |         |         |
  |         |         |         +-err_msg
  |         |         |         |
  |         |         |         +-lose_mg
  |         |         |
  |         |         +-lose_mg
  |         |
  |         +-err_msg-+-tx_msg
  |         |         |
  |         |         +-err_msg
  |         |         |
  |         |         +-lose_mg
  |         |
  |         +-lose_mg-+-tx_msg
  |                   |
  |                   +-err_msg
  |                   |
  |                   +-lose_mg
  |
  +-err_msg-+-tx_msg
  |         |
  |         +-err_msg
  |         |
  |         +-lose_mg
  |
  +-lose_mg

Search for all solutions complete -
21 valid sequences end in cycles.
Depth sufficent to find all cycles.
```

11 POTENTIAL DEVELOPMENTS FROM PETRI NET THEORY

11.1 A Packet Petri Net for Modelling Layered Protocols

A packet Petri net is an extension of the numeric Petri net, each token containing a list of integers. An application can be seen immediately to the modelling of communications protocols, the packet-token can represent a message of the protocol under analysis [BILL88]. The "TRAV" analysis tool, and "Needle" specification language (Appendix-F.11) could be extended to describe packet Petri nets, due to the excellent list processing ability of Prolog.

11.2 The Petri Net as a Powerful Computer Architecture

Define a Binary Petri Net as a Petri net where each token is a 'bit' carrying '0' or '1'. A physical implementation of a Binary Petri Net would be a very useful computer architecture. It could serve as the hardware host for implementing a 'virtual' packet Petri net architecture. A packet token in the virtual net will be implemented by a stream of tokens in the physical net. This is possible because the Binary Net is capable of parsing the stream, identifying its head and tail tokens.

Although put forward here as a vague speculation, current simulation tools [AJM086] are capable of analysing such a configuration. The simulation would include a formal specification of the token stream parsing mechanism. Given an appropriate benchmark application, quantitative comparisons could be made for two choices of host hardware; Binary Petri net and 16-bit microcomputer.

11.3 A Petri Net Parses its own Language

Figure 11.1 shows a numerical Petri net capable of parsing its own specification language to build Petri net models. Indeed it would construct another copy of itself when supplied with the specification. By distributing many parsing modules in a hierarchical tree, a very interesting Petri net machine could be constructed.

The constructor builds the Petri net with raw materials from resource pools, **place_pool**, **trans_pool**, etc. A stream of program instructions arrive at the **IN** port of the constructor. The **IN** port is not shown, but it is implied that it has an arc going to every transition in the diagram. Each transition has a **get(IN,X)** arc where **X** is the numeric code of a statement in the instruction set. The instruction stream is parsed by the illustrated network, and a Petri net is built at the **construction_site**. The constructed net is stored as lists, **place_list**, **trans_list**, etc.

During parsing, context places **place_context** and **trans_context** store the name of the current place or transition being constructed. The network propagates tokens which carry integer lists. For example the **source_name** transition sends a token **[tr,source]** to the **from_list** place.

An extension to the parser **Initial Marking Parser** (Figure 11.2) supplies the initial pattern of tokens into the constructed Petri net. Each initial token packet is constructed as an integer list held in the **build_token** place, by the instruction sequence:

begin_token, integer, integer, ... end_token

The begin and end token instructions can be represented by brackets **[** and **]**. On receiving the **end_token** instruction, the token is positioned in the **place_list** at the relevant location defined by **place_context**. Figure 11.3 illustrates the Petri net constructed from the following instruction sequence:

```
INSTRUCTION SEQUENCE

  module M
    place A          /* Place A initially contains one
      [ 1 2 3 ]      /* token of three integers
    place B
    transition C     /* Transition connects A to B
      from A
      to B
  module_end
```

The parser is a numeric Petri net, therefore the instruction set can be encoded numerically as follows.

CODE	INSTRUCTION
0	place
1	transition
2	from
3	to
4	get
5	put
6	module
7	end-module
8	**[** begin-token
9	**]** end-token

References relating to network architectures with the potential for self-programming are:

[Valk78] describes "Self-modifying nets, a natural extension of Petri nets". [BURT84,BURT88] describes virtual tree machines. [AGHA86] describes actors, a hierarchical model of concurrent computation.

Figure 11.1 Petri Net of A Petri Net Constructor

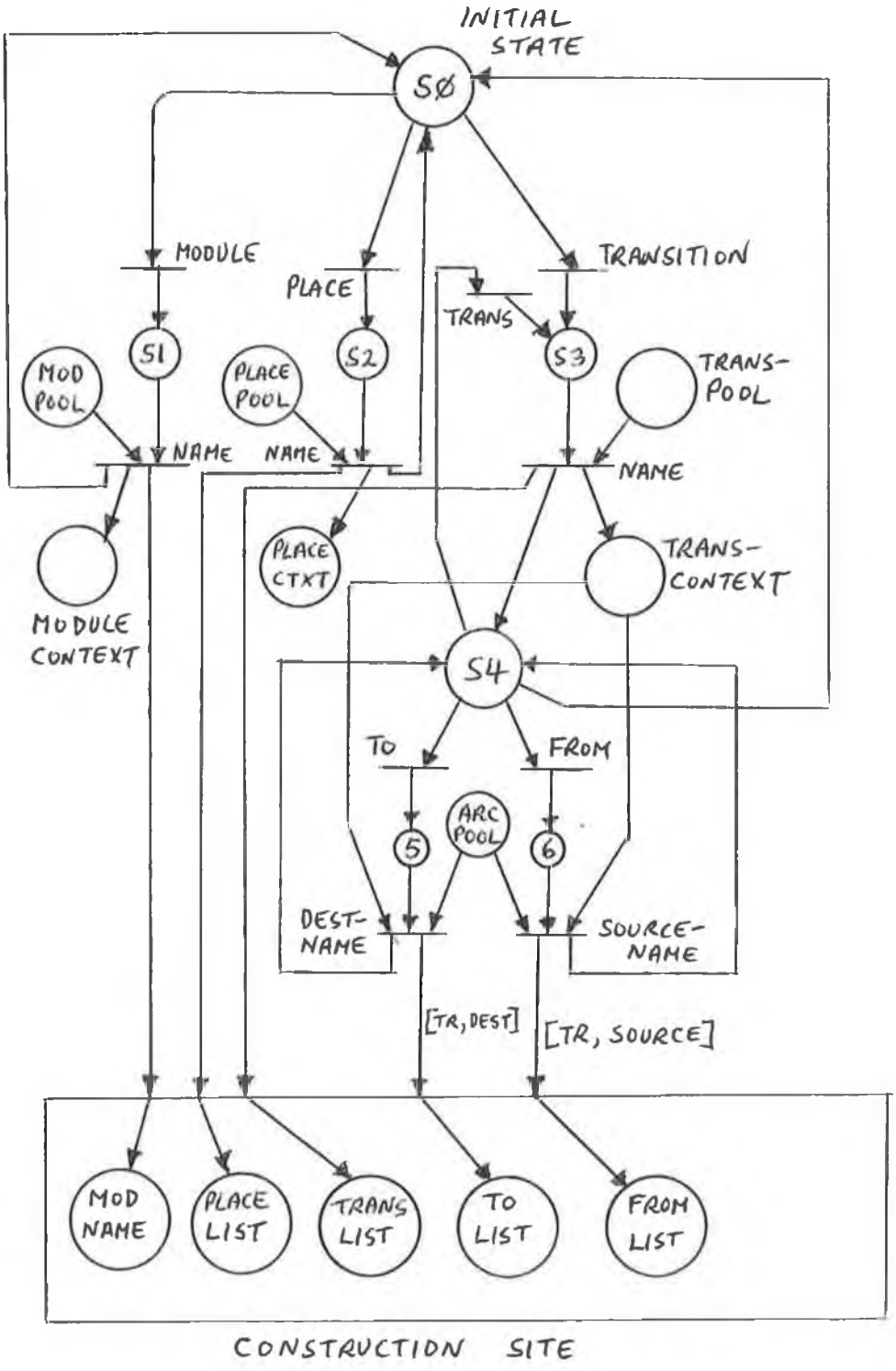


Figure 11.2 Initial Marking Parser

(Extension to Petri Net Constructor)

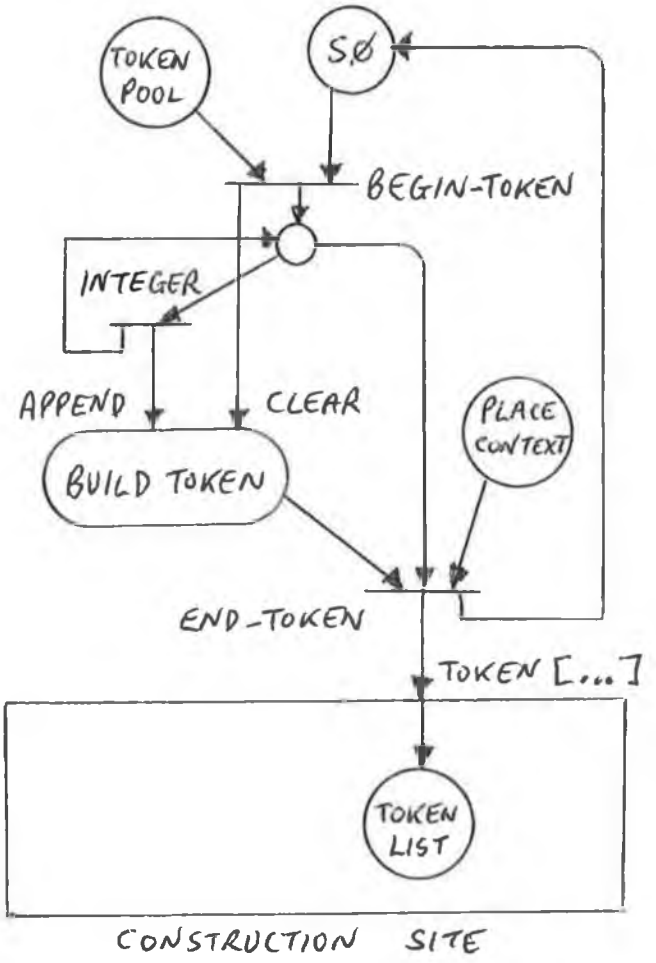
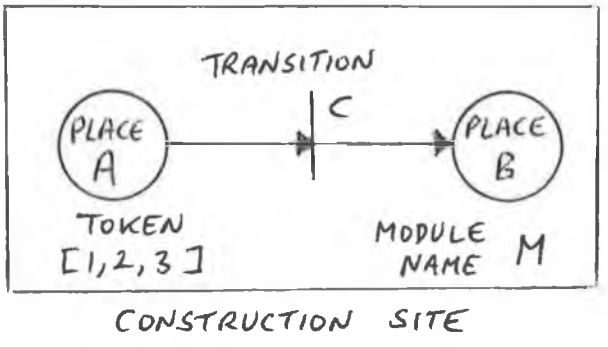


Figure 11.3 Constructed Petri Net

(FROM PROGRAM, PAGE 65)



11.4 Neural Petri Nets Capable of Learning

Artificial neural networks have been demonstrated to be capable of "learning", defined as use of feedback from the task in hand, to make a positive behavior adjustment. [LOON88] decribes a new type of neural network, obtained by extending a Petri net with "Fuzzy" valued rules and tokens. The Petri net having relevant interconnections only, is an advancement on previous neural networks which relied on mass interconnection between layers.

11.5 A Time Reversible Specification Language

The Needle specification language is time reversible. This means that a language translation exists which causes the specified model to run backwards in simulation time. "Running backwards" refers to the reachability analysis algorithm, which will now generate a reachability tree of possible pasts, instead of possible futures.

The required translation of Needle statements is:

IN <---> OUT

TO <---> FROM

GET <---> PUT

Graphically this means that all arrows in the specification reverse their direction, and input ports are exchanged with output ports.

[LOON88] in "Fuzzy Petri Nets for Rule-Based Decisionmaking" describes such a reversal. By reversing all arrows, he propagates the tokens backwards, thus causing the network to reason about its previous states.

The example below (Figure 11.4), shows time reversal of the Channel Module from the alternating bit protocol model (Appendix-E.3, Appendix-G.1).

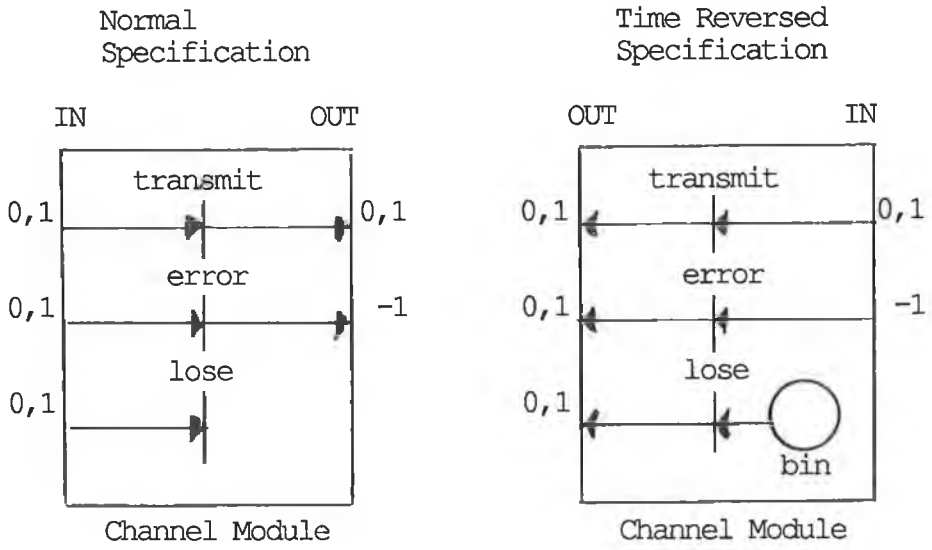
Looking at the **transmit** event in the channel module; its function is unchanged by time reversal. It still takes a '0' or '1' from the **IN** port and transmits the value to the **OUT** port.

Time reversal of the **error** event is a different matter. The reversed **error** event is now enabled by a '-1' (error) token, and has two possible futures; '0' goes out , or '1' goes out.

The **lose** event is a sink for all received tokens. Time reversal turns it into a spontaneous source of tokens. This would have disastrous consequences for computability of the reachability tree, which would quickly diverge due to all the spontaneous events. There is a simple solution, by adding a **bin** place, for disposing of lost tokens. This does not upset the original model, it simply acts as a means of counting the lost message events. With time reversal, the **bin** puts a ceiling on the number of spontaneous **lose** events, making the reachability analysis computable. This minor modification gives us a protocol model which we can run backwards in time, to predict its previous states.

To ensure that all application models are time reversible, some restrictions would have to be placed on the Needle specification language. The model may have to be built from **get** and **put** arcs only. Omitting **to** and **from** arcs, because they would propagate integer tokens with unknown values. Although if a Fuzzy Petri net were used, [LOON88], it would be capable of processing tokens with uncertain value.

Figure 11.4 Time Reversible Specification



11.6 Quantum Mechanics of Finite State Machines

This section discusses common concepts between two seldom related disciplines, quantum mechanics, and Petri net theory. The obvious connection is that both disciplines use directed graphs for descriptive purposes. [HASS89] gives a good example of a complex directed graph, the Feynman diagram of a high energy particle which comes into existance for a short time period, according to the Heisenberg uncertainty principle.

The second connection is that both describe systems which contain a discrete number of states, e.g. the Baryon octet classifies eight distinct states of a fundamental particle. See also Isospin and SU(3) Symmetry [Sakata '56],[Feynman, Lecture Notes III, Quantum Mechanics].

The third connection is that both disciplines provide mathematical methods for reasoning about timed events and probability. Petri net theory applies classical concepts of probability and time, for example to simulate the performance of a computer operating system [CHAN89].

Quantum reasoning about time and probability portrays a different reality. For example [Gell-Mann & Pais] the model for the transition of the K0-meson into its antiparticle. In simplistic terms, it could be said that the K0-meson and its antipartical are two states of a Finite State Machine. The transition from one state to the other has a particularly unusual time dependent formula, due to the interference of probability amplitudes.

The similarities between the two disciplines could be productive, if it is possible to make meaningful cross-application of their mathematical methods.

Transferring Petri net mathematics into particle physics, it would be possible to describe a Feynmann diagram using a Petri net specification language, because both are directed graphs. In both cases there is an operational significance in reversing the directions of all the arrows; CPT invariance, and backward execution.

Transferring the laws of quantum mechanics into a Petri net, every place would contain a probability amplitude instead of a token ! A Hamiltonian operator would be required, to describe how the system evolves from one state into another. This is by analogy with the mechanics of a half-spin particle in a magnetic field. Petri nets and spin mechanics both apply matrix transformations to a state vector. The Petri net uses an incidence matrix, quantum mechanics uses the Pauli spin matrices. Coincidentally, this text is being projected to the screen by a beam of half-spin particles in a magnetic field.

Analogy can also be drawn between a physical system, and a reachability analysis tool, otherwise reachability analysis wouldn't be of any use! The analysis tool pursues all possible branches of the reachability tree, but quickly consumes the available computing resources, a severe case of the finite-state explosion problem.

A physical system pursues all possible branches of its reachability tree, but is successful in managing its resources at a local scale in physical space. The resources are allocated in packets, called particles. The resource management method applies a probability amplitude to each branch of the reachability tree, and ensures that the sum of probabilities over alternative branches is unity. This is demonstrated by the Young's slits optical interference experiment. There are two alternative events, the choices of which slit the photon will pass through. The wave equations describe that both events are investigated concurrently. By analogy therefore, the wave equations of a photon describe the perfect algorithm for concurrent analysis of the reachable states of a system.

.oOo.

APPENDIX - A

STANDARD PETRI NET PLACE TYPES

Contents:

Reference: [MCAL87], Chapter 4.

A.1 CLASSIFICATION OF PLACE TYPES

A set of standard Petri net places is supplied for building timed models. They provide simulation features such as time delays and branch decisions. They are called up via the specification language, and customised by supplying parameters.

The table below classifies place types by the following characteristics:

- COLLECT TOKEN, EMIT TOKEN or PASS TOKEN defines whether it is the function of the place to detect events, drive events or both.

- DIRECT PASS indicates that an input event directly causes an output event. STORED PASS defines that the place has an internal memory of previous events.

- A TIMED place has internal timers which determine when a token will be made available at the output.

- TRANSIENT OUTPUT means that an output token is available for only a momentary instant, and is lost if not collected by an event. In the case of STORED OUTPUT, the available output token is stored until an event enables it to be output.

- The output of a DETERMINATE place is a function of its past sequence of input and output events. A RANDOM place simulates unpredictable output by using randomised numbers.

- Whether the place makes use of SINGLE or MULTIPLE tokens. The two types QUEUE and CHANNEL which are specified as using multiple tokens have applications in communications modelling.

Table A.1: Functional Characteristics of Timed Petri Net Places.

PLACE TYPE / CHARACTERISTIC	ADMIT	STATE	TIMER	DELAY	TRAIN	METER	QUEUE	CHANNEL
COLLECT TOKEN	.	.	.	.	.	Y	.	.
EMIT TOKEN	.	.	.	.	Y	.	.	.
PASS TOKEN	Y	Y	Y	Y	.	.	Y	Y
DIRECT PASS	Y	Y	.	.	NA	NA	.	.
STORED PASS	.	.	Y	Y	NA	NA	Y	Y
UNTIMED	Y	Y	.	.	.	.	Y	.
TIMED	Op	Op	Y	Y	Y	Y	.	Y
TRANSIENT OUT	Y	.	Y	.	Y	NA	.	Y
STORED OUTPUT	.	Y	.	Y	.	NA	Y	.
DETERMINATE	Y	Y	Y	Y	Y	NA	Y	Y
RANDOM	Op	Op	Op	Op	Op	NA	.	Op
SINGLE TOKEN	Y	Y	Y	Y	NA	NA	.	.
MULTI TOKEN	.	.	.	.	NA	NA	Y	Y

Key: . = No
Y = Yes
NA = Not Applicable
Op = Option

A.2 SPECIFICATION OF THE METER PLACE

Place types are called up from the specification language. The METER place is only one of the available types, and is described here to illustrate operation of the simulator.

Metering places are declared as part of the application model being simulated. They control collection of simulation statistics and provide automated generation of performance analysis reports.

The METER is a module with two input ports; INPUT and ALTERNATE-INPUT. It has no output events, but simply records and reports on monitored input events. Instances of the METER are declared as required, to collect specific statistics from the Petri net model. They plug into the model via connecting arcs from the transitions being monitored. Qualifying parameters COUNTER, STOPWATCH, UPTIME and EVENTS select the mode of recording according to the application.

The four recording modes are described below:

Counter mode:

During simulation the meter monitors a particular event and counts the number of times it occurs. The second input of the meter allows a series of counts to be stored and averaged. On completion of the run, the meter generates reports of minimum, average and maximum recorded counts.

INPUT:	Count	- increment event counter
ALT INPUT:	Restart	- store count and restart

Stopwatch mode:

Measures the simulated time interval between two specified events. A series of measurements may be recorded. The reports give minimum, average and maximum time measurements.

INPUT: Start watch

ALT INPUT: Stop watch

Equipment Uptime mode:

Used for performance simulation of manufacturing plants, or computer networks which contain equipment prone to failure. The meter monitors a process, measuring the active interval (uptime), and inactive interval (downtime). It reports appropriate statistics, for example a count of the number of equipment failures which exceed the permitted downtime period.

INPUT: Up - Start of uptime (Equipment recovery event)

ALT INPUT: Down - Start of downtime (Equipment failure event)

Event Recording Mode:

Used to capture the trace of an event sequence, e.g. leading to an exception condition. The simulator records the trace in circular trace buffer. An effective method of letting the simulation run randomly for a long time, only capturing information, when a particular events occur.

INPUT: Enable recording of events.

ALT INPUT: Stop recording, and store trace buffer.

APPENDIX - B

PERFORMANCE SIMULATION OF ALTERNATING BIT PROTOCOL

Contents:

Reference: [MCAL87], Chapter 6.

Introduction

This is a timed model of the alternating bit protocol, simulating transmission on an unreliable communications channel.

Listing B.1 is the model specification processed by the simulator. The simulation also requires some 'C' language modules (Appendix-C).

Listings B.2 to B.5 are the output reports of the simulator. B.5 is suitable for generating a performance graph of the protocol.

B.1 SPECIFICATION OF ALTERNATING BIT MODEL

```
SIMULATION  PACKET-1.SIM          Sending and Receiving Nodes
UNIT    ms                /* millisec time unit.
MAXTIME 20000 ms          /* Max simulation time
REPORT                    /* Report options..
  TRACE                      /* Listing B.2
  OCCUR                      /* Listing B.3
  MODULE                     /* Listing B.4
PLACES
  AHOST S-host  3 ms    50 packets      /* Sending host
  BHOST R-host  5 ms                    /* Receiving host

  OUTNODE       S-node  50 ms   /* Retransmit timeout
  INNODE        R-node

  CHANNEL Channel-to-R    10 ms       /* Transmission delay
  ADMIT   Intact-to-R     PROB 80%    /* Msg rxd valid checksum

  CHANNEL Channel-to-S    10 ms       /* Transmission delay
  ADMIT   Intact-to-S     PROB 80%    /* Msg rxd valid checksum

  METER transmit-watch  STOPWATCH       /* Measure transmission
  METER response-watch  STOPWATCH       /* Measure turn around
EVENTS              /* The EVENTS specify the interconnection of
  (continued...)    /* of places.  EVENT = a Petri net transition.
                    /* Order in which they are declared is irrelevant.
```

B.1 SPECIFICATION (continued)

```
EVENTS
 EVENT S-host-emit-data          /* Pass data packet from host to node
    FROM S-host                     /* (Arc from source place)
      TO S-node                     /* (Arc to destination place)
          TO transmit-watch
          TO response-watch         /* Start the two stopwatches

 EVENT S-tx-data                 /* Node transmits data to channel
    FROM ~S-node
      TO Channel-to-R

 EVENT R-arrive-data
    FROM Channel-to-R            /* End of transmission delay
      TO Intact-to-R             /* Now introduce random signal errors

 EVENT R-rx-bad-pkt              /* Data packet lost or damaged
    FROM ~Intact-to-R

 EVENT R-rx-good-data            /* Data packet received intact
    FROM Intact-to-R
      TO ~R-node

 EVENT R-tx-ack                  /* Receive node transmits ACKnowledge
    FROM ~R-node
      TO Channel-to-S

 EVENT S-arrive-ack
    FROM Channel-to-S            /* End of transmission delay
      TO Intact-to-S             /* now introduce random signal errors

 EVENT S-rx-bad-pkt              /* ACKnowledge lost or damaged
    FROM ~Intact-to-S

 EVENT S-rx-good-ack             /* ACK received intact
    FROM Intact-to-S
      TO ~S-node

 EVENT S-host-collect-ack        /* Node passes ACK to (Sending) host
    FROM S-node
      TO S-host
          TO ~response-watch           /* Stop the stopwatch

 EVENT R-host-collect-data       /* Node passes data to receiving host
    FROM R-node
      TO R-host
          TO ~transmit-watch           /* Stop the stopwatch

 EVENT R-host-emit-ack           /* Receiving host passes ACK to node
    FROM R-host
      TO R-node
END
```

B.2 TRACE REPORT OF EVENT SEQUENCE

```
------------------------------------------------------------

SIMULATION  PACKET-1.SIM        Sending and Receiving Nodes

----------   TRACE REPORT:   Sequence of Events   ------------

Time (+increment)
Unit: ms
                    /* Event cycle for 1st packet
3   (+3)                  S-host-emit-data
3   (+0)                  S-tx-data
13   (+10)                R-arrive-data
13   (+0)                 R-rx-good-data
13   (+0)                 R-host-collect-data
13   (+0)                 R-host-emit-ack
13   (+0)                 R-tx-ack
23   (+10)                S-arrive-ack
23   (+0)                 S-rx-good-ack
23   (+0)                 S-host-collect-ack
                    /* Event cycle for 2nd packet
26   (+3)                 S-host-emit-data
26   (+0)                 S-tx-data
-                         -
-                         -
```

B.3 EVENT OCCURANCE REPORT

```
------------------------------------------------------------

---------------     Event Occurance Report      ---------------

OCCURANCES       TRANSITION

    50           S-host-emit-data
    78           S-tx-data
    78           R-arrive-data
    16           R-rx-bad-pkt
    62           R-rx-good-data
    62           R-tx-ack
    62           S-arrive-ack
    12           S-rx-bad-pkt
    50           S-rx-good-ack
    50           S-host-collect-ack
    50           R-host-collect-data
    50           R-host-emit-ack

------------------------------------------------------------
```

B.4 MODULE REPORT

```
-----------------------------------------------------------------

SIMULATION  PACKET-1.SIM          Sending and Receiving Nodes

-----  MODULE REPORT:   Place Definitions and Statistics  -------


MODULE REPORTS

  SENDING HOST   S-host   Send interval: 3 ms  Pkts/session: 50
                 Packets sent in new session:  50

  RECEIVING HOST   R-host   Session length: 50
                 Ready for next session


  OUT-NODE  S-node  Retransmit timeout: 50 ms
    Tx 1st attempts: 50     Tx retries: 28

  IN-NODE  R-node
    Rx accept count: 50     Rx reject count: 12
    Frames received: 62     Acks transmitted: 62


  CHANNEL Channel-to-R   Delay: 10 ms


  ADMIT Intact-to-R PROB 80 %   Count 62 successes, 16 failures
                                Success frequency   79.49 %

  CHANNEL Channel-to-S   Delay: 10 ms

  ADMIT Intact-to-S PROB 80 %   Count 50 successes, 12 failures
                                Success frequency   80.65 %


  METER transmit-watch STOPWATCH          Number of samples: 50
        Average time     23.00 ms
        Minimum time     10    ms
        Maximum time    160    ms

  METER response-watch STOPWATCH          Number of samples: 50
        Average time     48.00 ms
        Minimum time     20    ms
        Maximum time    170    ms

-----------------------------------------------------------------
```

B.5 ANALYSIS REPORT

(For generating a performance graph. X = Frequency, Y = Average-T)

SIMULATION PACKET-2.SIM Sending and Receiving Nodes

--------- ANALYSIS REPORT: Repeated Simulations ---------

Controlled Probability of Packet Transmission

Run No.	Channel-to-R			Channel-to-S		
	Prob	Frequen	Samp	Prob	Frequen	Samp
1	40	42.01	288	40	41.32	121
2	50	45.29	223	50	49.50	101
3	60	61.54	169	60	48.08	104
4	70	71.29	101	70	69.44	72
5	80	85.33	75	80	78.13	64
6	90	94.74	57	90	92.59	54
7	100	100.00	50	100	100.00	50

Resultant Measurement of Response Times

Run No.	transmit-watch				response-watch			
	AverageT	MinTim	MaxTim	Samp	AverageT	MinTim	MaxTim	Samp
1	92.00	10	610	50	258.00	20	920	50
2	67.00	10	510	50	193.00	20	1070	50
3	45.00	10	310	50	139.00	20	670	50
4	30.00	10	110	50	71.00	20	370	50
5	17.00	10	110	50	45.00	20	220	50
6	13.00	10	110	50	27.00	20	170	50
7	10.00	10	10	50	20.00	20	20	50

PROTOCOL SIMULATION 'C' LANGUAGE MODULES

Contents: C.1 Data Structure File SIMUL.H

C.2 Sending Host Module SHOST.C

Reference: [MCAL87], Chapter 6.2

Introduction

These listings illustrate 'C' language modules required for the alternating bit protocol model. SIMUL.H is an include file common to all modules, it defines the program constants and data structures used by the simulator. SHOST.C is one of four application modules used by the specification in appendix-B. (The other three RHOST, OUTNODE, INNODE follow a similar program structure.)

C.1 'C' LANGUAGE INCLUDE FILE SIMUL.H

```
/*****************************************************************/
/*      11/Sep/86      SIMUL.H   Constants and data structures  */

#define FALSE   0       /* Boolean logic */
#define TRUE    -1

#define EMIT    -1     /* Primary interface ports, output and input */
#define COLLECT 1
#define ALTEMIT -2      /* Secondary (ALTernate) interface ports */
#define ALTCOLLECT 2

#define PERCENT '\045'  /* the ASCII percent character for reports */
#define FOREVER 0x7fff  /* Max signed 16 bit integer */
#define pmax 20         /* Max number of Petri net places */
#define tmax 20         /* Max number of Petri net transitions */
#define MAXLINE 81      /* 80 character line plus '0' terminator */
#define name_size 20    /* Length of names of places/transitions */
#define var_num 8       /* Number of local variables in a place */
#define attr_num 4     /* No. of attributes passed by a transition */
```

```
/* The data structure for each place, the model has array of these */

typedef struct Place_struc {
        char    name[name_size];        /* the name of the place */
        int     ptype;   /* selects type of predefined place module */
                         /* and thus pointers of 6 functions below */
        int     sleep;     /* How long before port outputs a token  */
        int     altsleep;       /* How long before alternate port.. */
        int     last_input;     /* Store value which was last input */
        int     last_altinput;  /* Store value on alternate input */
        int     last_output;   /* Store value which was last output */
        int     last_altoutput;   /* Store last alternate output */
        int     countin;        /* Statistics, count input events */
        int     altcountin;     /* Count of alternate input events */
        int     countout;       /* Count of events on output port */
        int     altcountout;    /* Count of alternate output events */
        int     v[var_num];    /* Local variables for place process */
           /* 6 function pointers make calls to predefined modules */
        int     (*elapse)();  /* Ptr to function which elapses time */
        int     (*input)();     /* Pointer to func. does input event */
        int     (*altinput)(); /* Function does event on alt input */
        int     (*output)();   /* Func. does output port event */
        int     (*altoutput)();  /* Function does alternate output */
        int     (*report)();  /* Func. prints report for this ptype */
} Place_struc, *Place_ptr;


/* The structure for each transition, the model has array of these */

typedef struct Trans_struc {
        char    name[name_size];
        char    ttype[name_size];
        int     a[attr_num];    /* The msg packet passed from src
                                   places to destination places */
        int     fireable;  /* Firable if source places have tokens */
        int     countfire; /* Statistics of number of firing events */
} Trans_struc, *Trans_ptr;


/* The structure for each specific type of predefined place */
                                        /* Used in PTYPE.C */
typedef struct Pl_type_struc {
        char    type_name[name_size];
        void    (*place_init)();
} Pl_type_struc, *Pl_type_ptr;


/*****************************************************************/
```

C.2 SENDING HOST 'C' LANGUAGE MODULE

```
/*****************************************************************/

/*      SHOST.C

        Sending HOST.           Transmits messages at specified interval
                                Specified number of packets in session.

        6 Mar 1987
*/


#include        <stdio.h>
#include        "menu.h"
#include        "simul.h"       /* Constants and data structures */

extern Place_struc p[];         /* Array of structures for places */
extern Trans_struc t[];         /* Array of structs for transitions */
extern char t_unit[];           /* Unit of time eg. "sec" or "hour" */

        /* Send interval */
#define delay   p[pl].v[0]      /* v[0..7] are internal variables */

        /* Total num of msgs to send */
#define seslen  p[pl].v[1]

    /* Dummy msg contents is send counter  e.g. msg sequence 5 2 3 4 5
       scnt in first msg is size, in subsequent msgs is counter */
#define scnt    p[pl].v[2]

        /* Contents of message packet , 3 integer values */

#define msg_type        t[tn].a[0]      /* Message packet */
#define msg_seq         t[tn].a[1]      /* Sequence number */
#define msg_data        t[tn].a[2]      /* 3rd word of packet */

                        /* Functions in this module, see below */
void    shost_elapse(),
        shost_send(),
        shost_receive(),
        shost_report();
```

```
/* Function initialise the data structures for this place type */

extern void shost_init(pl, cmd)
int pl;
char cmd[];
{
char s[MAXLINE];

p[pl].elapse = shost_elapse;     /* Assign function ptrs */
p[pl].output = shost_send;
p[pl].input = shost_receive;
p[pl].report = shost_report;

     /* Parse 1 line of the specification file for parameter values */
     /* Command format    "A_HOST name interval tunit session_len" */

sscanf(cmd,"%s %s %d %s %d",s ,p[pl].name, &delay, s, &seslen);
scnt = 0;
               /* This module is unusual in that it is capable */
               /* of initiating its first output i.e. finite sleep */
               /* is defined at init.  */
p[pl].sleep = delay;     /* Interval to first msg */

}

/*-----------------------------------------*/

/* Function simulates passage of time,
   "elapse" is the number of time units to elapse,
   as supplied by the simulator kernal at each step */

void shost_elapse(pl,elapse)
int pl,elapse;          /* pl is index to array of place structures */
{
if (p[pl].sleep < FOREVER)
  {  p[pl].sleep = p[pl].sleep - elapse;
     p[pl].sleep = max(0,p[pl].sleep);
  }
}
```

```
/*----------------------------------------*/

/* Function outputs a message on output port */
/* Message represents transmit of protocol data packet to S-node */

void shost_send(pl,tn)
int pl,tn;
{
 ++scnt;         /* Count number of packets sent by Sending host */
 if (scnt==1)  msg_data = seslen;
                          /* First msg tells total num of packets */
   else msg_data = scnt;   /* Put scnt in packet as dummy data */

        p[pl].sleep = FOREVER;
                          /* Wait till msg is acked by our node */
}


/*----------------------------------------*/

/* Function inputs a message on input port */
/* Message represents receipt of acknowledge packet from S-node */

void shost_receive(pl,tn)
int pl,tn;
{
 if (scnt<seslen)
        p[pl].sleep = delay;     /* interval to next msg */
   else
        p[pl].sleep = FOREVER;          /* No more messages */
}


/*----------------------------------------*/

/* At end of simulation function prints statistics to report file */

void shost_report(pl,dst_ptr)
int pl;
FILE *dst_ptr;
{
fprintf(dst_ptr,
     "  SENDING HOST   %s   Send interval: %d %s  Pkts/session: %d\n"
        ,p[pl].name, delay, t_unit, seslen);
fprintf(dst_ptr,
        "                    Packets sent in new session:  %d\n", scnt);
}

/*----------------------------------------------------------------*/
```

APPENDIX - D

SDL Specification of Alternating Bit Protocol

Contents:

D.1 SDL Specification of Interacting Processes

The examples and listings given in this appendix are from [CAVA87], to which reference should be made for a more complete description. The protocol modelled is the "ALTERNATING BIT PROTOCOL", a classic protocol used to provide reliable message flow between sender and receiver via an unreliable channel. The model implements an interactive simulation of the protocol. It is written in "SDL", the Specification and Description Language standardised by the CCITT. The specification is listed in full below.

The specification program defines three interacting processes:

sender, receiver and channel.

The function of the system is to model transmission of data from sender to receiver via the channel. The channel introduces errors in the data flow. The alternating bit protocol is designed to recover from errors. The protocol is implemented by the sender and receiver processes. The SDL specification defines the processes as

state machines which communicate by input and output of messages. The model runs interactively, and errors are introduced on the channel by "yes/no" input from the user. The model is a state transition machine, states are identified by the keyword "STATE", transitions by the keyword "NEXTSTATE".

D.2 Internal States of Each Process

Sender has 4 states:

idle0 - pending data from input queue,
sequence counter is 0

waitam0 - pending ack with sequence count 0

idle1 - pending data from input queue,
sequence counter is 1

waitam1 - pending ack with sequence count 1

Receiver has 2 states:

waitdm0 - pending data msg with sequence count 0

waitdm1 - pending data msg with sequence count 1

Channel has 5 states:

empty - channel contains no message

content_dm0 - data message with sequence count 0

content_dm1 - data message with sequence count 1

content_am0 - ack message with sequence count 0

content_am1 - ack message with sequence count 1

D.3 Message Names and Terminology

The following messages are used by the SDL program:

User messages (between user and program):

- dm - data message to send
- dm_zero - received data msg
- dm_one - received data msg
- clear - yes/no decision to generate error

Protocol messages (between processes):

- dm0, dm1 - data messages
- am0, am1 - acknowledge messages
- error - generated error message

Terminology used in constructing names of states or messages:

- idle - do nothing until there is data to transmit
- wait - wait to receive a message from the channel
- 0,1 - modulus 2 sequence counter
- dm - data message
- am - acknowledge message
- content - a message passing through the channel

```
/***  SDL SPECIFICATION OF ALTERNATING BIT PROTOCOL  ***/

/*** Listings from [CAVA87] with minor modifications ***/

/* D.4    SENDER PROCESS SPECIFICATION

PROCESS sender;

STATE idle0;

  INPUT dm;                        /* Accept message
    OUTPUT dm0 TO channel;         /* transmit with sequence
    NEXTSTATE waitam0;             /* count of 0

STATE waitam0;                     /* Expecting ack msg 0

  INPUT am1;                       /* but get ack msg 1
    OUTPUT dm0 TO channel;         /* retransmit data msg
    NEXTSTATE waitam0;

  INPUT am0;                       /* Get correct ack msg
    NEXTSTATE idle1;

  INPUT error;                     /* Get damaged msg
    OUTPUT dm0 TO channel;
    NEXTSTATE waitam0;

STATE idle1;

  INPUT dm;                        /* Accept message
    OUTPUT dm1 TO channel          /* transmit with sequence
    NEXTSTATE waitam1              /* count of 1

STATE waitam1;                     /* Expecting ack msg 1

  INPUT am0;                       /* but get ack msg 0
    OUTPUT dm1 TO channel;         /* retransmit data msg
    NEXTSTATE waitam1;

  INPUT am1;                       /* Get correct ack msg
    NEXTSTATE idle0;

  INPUT error;                     /* Get damaged msg
    OUTPUT dm1 TO channel;
    NEXTSTATE waitam1;

ENDPROCESS;
```

/* D.5 RECEIVER PROCESS SPECIFICATION

```
PROCESS receiver;

STATE waitdm0;                          /* Expecting data msg 0

  INPUT dm1;                            /* but get data msg 1
    OUTPUT am1 TO channel;
    NEXTSTATE waitdm0;

  INPUT dm0;                            /* Get correct msg
    OUTPUT am0 TO channel;              /* acknowledge it
    OUTPUT dm_zero;                     /* and forward to user
    NEXTSTATE waitdm1;

  INPUT error;                          /* Get damaged msg
    OUTPUT am1 TO channel;
    NEXTSTATE waitdm0;

STATE waitdm1;                          /* Expecting data msg 1

  INPUT dm0;                            /* As above for waitdm0
    OUTPUT am0 TO channel;              /* but the zero's and
    NEXTSTATE waitdm1;                  /* one's are exchanged ...

  INPUT dm1;
    OUTPUT am1 TO channel;
    OUTPUT dm_one;
    NEXTSTATE waitdm0;

  INPUT error;
    OUTPUT am0 TO channel;
    NEXTSTATE waitdm1;

ENDPROCESS;
```

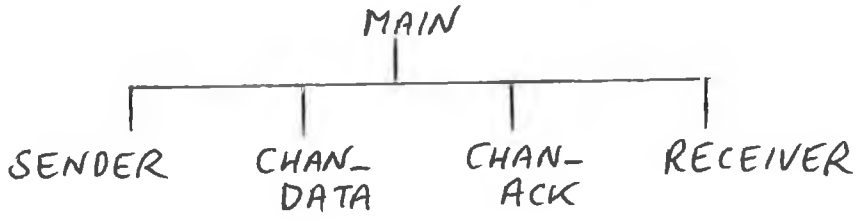
/* D.6 CHANNEL PROCESS SPECIFICATION

```
PROCESS channel;

STATE empty;                          /* Channel initially empty
  INPUT dm0;                            /* msg from sender
    NEXTSTATE content_dm0;                /* store the msg
  INPUT dm1;                            /* msg from sender
    NEXTSTATE content_dm1;
  INPUT am0;                            /* msg from receiver
    NEXTSTATE content_am0;
  INPUT am1;                            /* msg from receiver
    NEXTSTATE content_am1;

STATE content_dm0;                     /* Channel contains msg
  INPUT clear;                         /* decide to damage it
    DECISION 'error?';
      'yes' : OUTPUT error TO receiver;
         NEXTSTATE empty;
      'no' : OUTPUT dm0 TO receiver;
         NEXTSTATE empty;
    ENDDECISION;

STATE content_dm1;
  INPUT clear;                         /* decide to damage it
    DECISION 'error?';
      'yes' : OUTPUT error TO receiver;
         NEXTSTATE empty;
      'no' : OUTPUT dm1 TO receiver;
         NEXTSTATE empty;
    ENDDECISION;

STATE content_am0;
  INPUT clear;                         /* decide to damage it
    DECISION 'error?';
      'yes' : OUTPUT error TO receiver;
         NEXTSTATE empty;
      'no' : OUTPUT am0 TO receiver;
         NEXTSTATE empty;
    ENDDECISION;

STATE content_am1;
  INPUT clear;                         /* decide to damage it
    DECISION 'error?';
      'yes' : OUTPUT error TO receiver;
         NEXTSTATE empty;
      'no' : OUTPUT am1 TO receiver;
         NEXTSTATE empty;
    ENDDECISION;

ENDPROCESS;
```

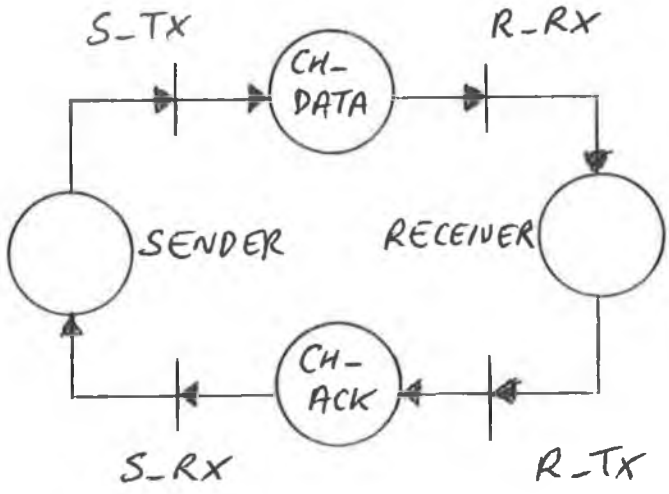
APPENDIX - E

SCHEMATIC SPECIFICATION OF ALTERNATING BIT PROTOCOL

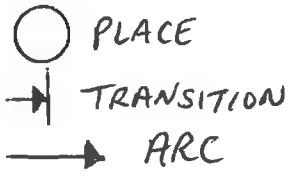
E.1 HIERARCHICAL MODEL



E.2 SCHEMATIC OF MAIN MODULE



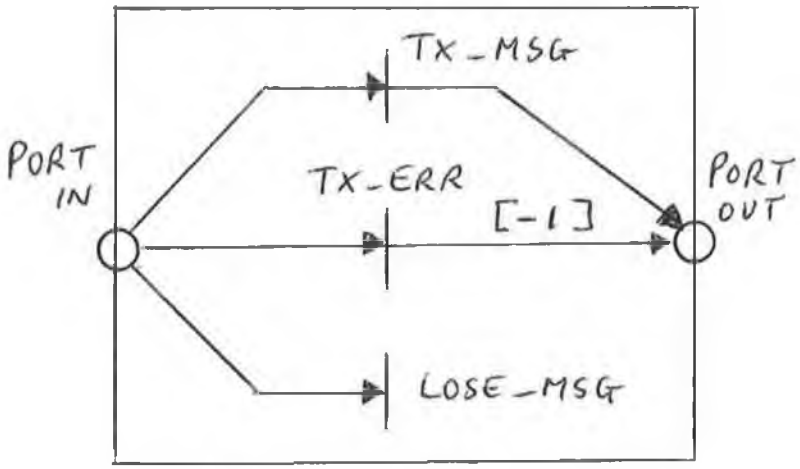
KEY:



APPENDIX - E

E.3

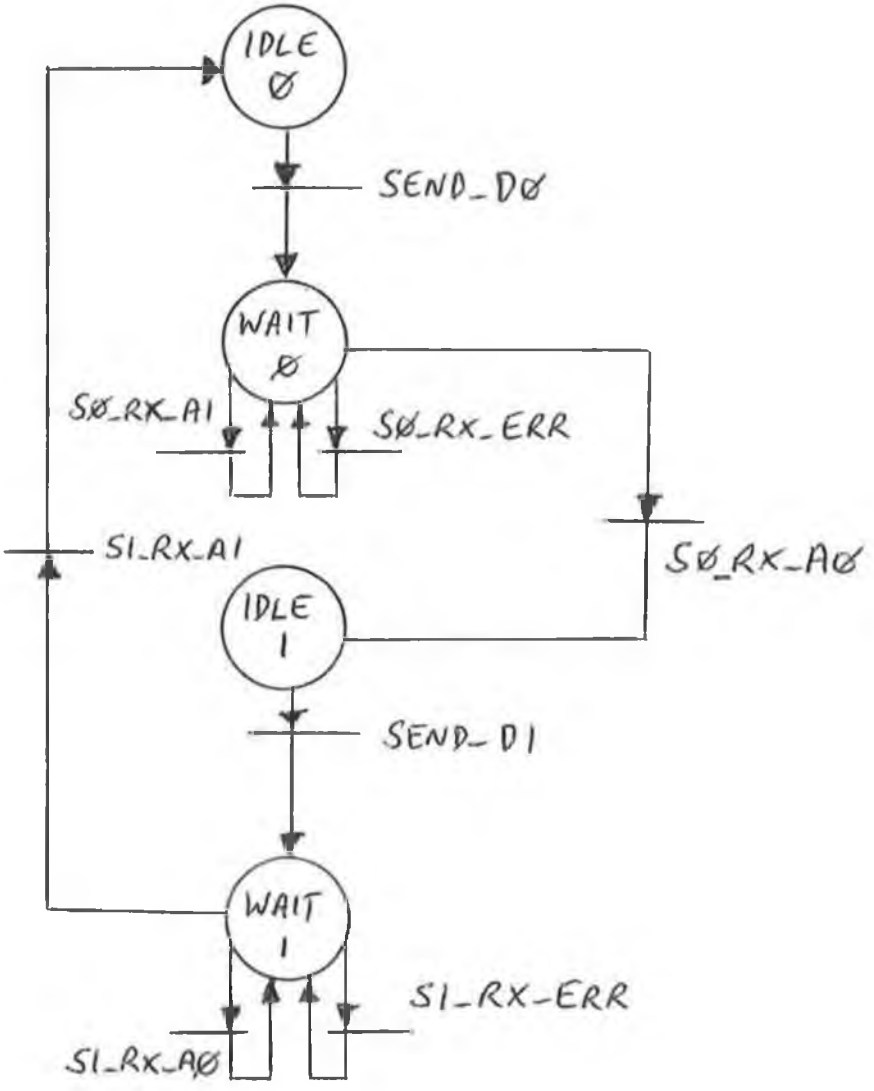
SCHEMATIC OF CHANNEL MODULE



TRANSITION TX_ERR CHANGES THE TOKEN VALUE TO -1.

E.4

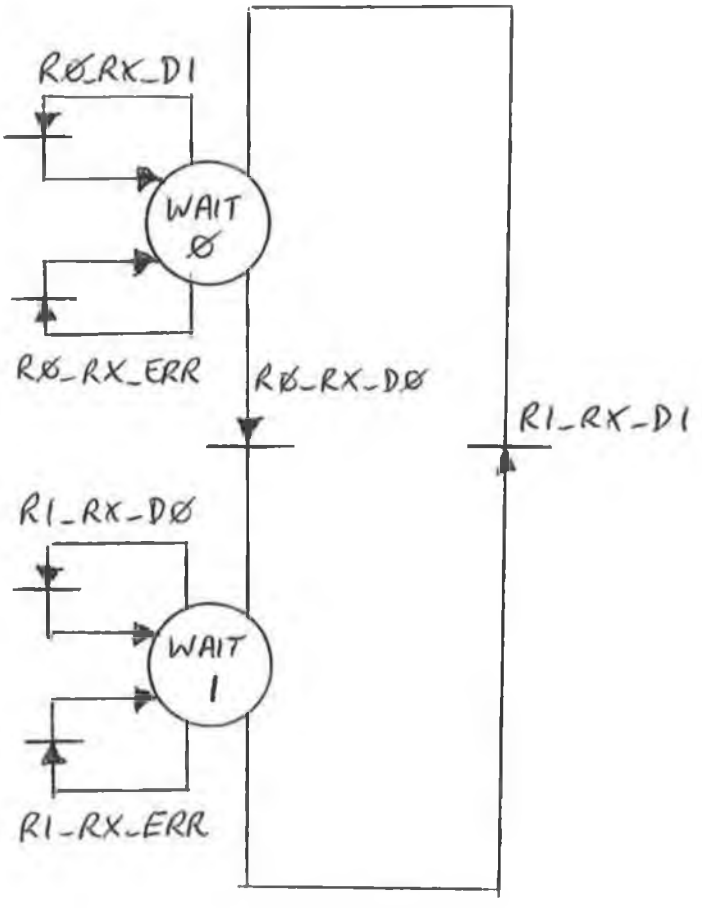
SCHEMATIC OF SENDER MODULE



EVERY TRANSITION HAS ARC(S)
TO INPUT OR OUTPUT PORTS,
OMITTED FOR CLARITY

E.5

SCHEMATIC OF RECEIVER MODULE



EVERY TRANSITION HAS ONE ARC TO INPUT PORT AND ONE ARC TO OUTPUT PORT, OMITTED FOR CLARITY

APPENDIX - F

"Needle" - A SPECIFICATION LANGUAGE FOR MODULAR STATE TRANSITION NETWORKS

Syntax Definition in Backus-Naur Format.

1. Introduction

"Needle" is a language for specifying state transition networks such as Finite State Machine and Petri net models. A modular numerical Petri net model provides the abstract basis for expressing and validating complex systems.

The language is highly parallel, composing a model from directed arcs as opposed to sequential statements. It may be suitable for future implementation on a highly parallel computer architecture.

The system specification satisfies the syntax defined below. The system is specified as a text file, in the form of a Prolog program. The Prolog compiler is Borland Turbo Prolog version 2.0 for the IBM Personal computer. The specification is compiled and validated by the network traversal program TRAV.PRO, also written in Prolog.

For comparison with the syntax definition of another language, see [JENS75] "Pascal User Manual and Report", p.110-115 Appendix-D Syntax.

Underlines	Key words in Prolog or Needle languages.
<>	A term expanded elsewhere.
::=	Indicates expansion of a term.
{}	A term occuring zero or more times
|	Indicates a selection from alternative terms

2. Top Level Structure of Specification Program

<specification> identifies the system specification held in a text file. Turbo Prolog requires declaration of predicates in a <predicates section>. The <clauses section> contains the system specification as a Petri net model.

<specification> ::= <predicates section> <clauses section>

<predicates section> ::= predicates
spec
conditions
<module predicates>
main

<module predicates> ::= { <module name> }

<clauses section> ::= clauses
<spec clause>
<conditions clause>
{ <module clause> }
<main clause>

3. The Specification Clause

The <spec clause> provides a single clause by which the analysis program can call up all the other clauses in the specification file.

```
<spec clause> ::=    spec :-
                       begin_spec,
                         conditions,
                         { <module name> , }
                         main,
                       end_spec.
```

4. Specifying Test Conditions

Test conditions are normally controlled interactively during analysis of the model. This section allows default test conditions to be defined. For further details on the commands refer to the User Instructions of the analysis program.

```
<conditions clause> ::= conditions :-
                            { <condition command> , } .

<condition command> ::= <search control command>
                      | <terminal condition command>
                      | <event control command>
                      | <display control command>
```

APPENDIX - F

<search control command> ::=

depth(<integer>)

| permit_loops(yes | no)

| first_result(yes | no)

<terminal condition command> ::=

initial (<token list> >

| end_option(cycle | state | deadlock)

| end_state(<token list>)

<event control command> ::=

avoid (<event list>)

| occur (<ordered event list>)

<display control command> ::=

show (<path list>)

| tree(yes | no)

| track(yes | no)

5. Module Declarations

Each module is written as a program block, bounded by the keywords module() and end(). A module defines a state transition network. The place() declaration will be used to declare instances of the module.

Module instances may be nested to various depths, and ultimately nested in the main module. Basic modules must be declared before compound modules of which they are a component.

Note that the specification language has placed scope limitations on ports, places and transitions, but not on module declarations. I.e. module declarations must have unique names, and cannot be nested.

```
<module clause> ::= <module name> :-
                        module( <module name> ) ,
                          { <port declaration> , }
                          { <place declaration> , }
                          { <transition declaration> , }
                        end( <module name> ).
```

6. Main Module Defines the Overall System.

The main module is the last block specified in the file. Its places declare instances of previously declared sub-modules. The main module may be declared with no ports since it is an isolated system.

APPENDIX - F

```
<main clause> ::= main :-
                      module( main ) ,
                        { <port declaration> , }
                        { <place declaration> },
                        { <transition declaration> },
                      end( main ).
```

7. Structure of a Module

Each module may contain a state transition network. The places and transitions are local to the module in which they are declared.

The transition network is defined by directed arcs. Arc connections are local to the module, with the exception of arcs which connect from the public deadlock place.

Communication beyond the module is via ports. A module may have any number of ports. Where modules have only one input or one output, the ports should be declared with the default names port(in) and port(out).

Arcs can only connect to ports of the local module, or to places within that module. By default, arcs from a place come from port(out) of that place. Arcs to a place go to port(in).

When places have multiple ports, the arcs will accept an extra parameter after the <place name> to explicitly specify the port.

APPENDIX - F

```
<port declaration> ::= port( <port name> )

<place declaration> ::= place( <place type> , <place name> )

<place type> ::= elementary | <module name>

<transition declaration> ::= transition( <transition name> ) ,
                                { <arc from> , }
                                { <arc to> , }

<arc from> ::= from( <source> )
             | get( <source> , <integer> )

<arc to> ::=  to ( <destination> )
            | put ( <destination> , <integer> )

<source> ::= <port name>                    /* of local module */
           | <place name>                         /* implicit */
           | <place name> , <port name>           /* explicit */
           | <public place>

<public place> ::= deadlock

<destination> ::= <port name>               /* of local module */
                | <place name>                    /* implicit */
                | <place name> , <portname>       /* explicit */
```

8. Numerical Extension to Petri Net

The get() and put() arcs provide numerical operations to the Petri net model. All arcs communicate tokens which contain an integer value. An arc get(<source>,<integer>) may enable a transition only if the token has the correct value. All input arcs must be enabled for a transition to fire. put() arcs emit a token with a specified integer value. to() arcs emit a token containing the maximum integer of all input arcs.

9. Extension to a Neural Network Model

It may be desirable in future to extend the the Petri net model to have more complex transition firing rules. This could be done by an extra field in the transition declaration. For example neural networks are capable of learning, by summing the input integers and emitting a value which is a non linear function of the total.

<transition declaration> ::=

transition(<transition name> , <firing rule>) ,

{ <arc from> , }

{ <arc to> , }

10. Syntax of Data Structures

Data structures are required as parameters to the test condition commands in (4.) above. They are used internally during analysis of the model, and are also displayed on result reports.

The square brackets [] are used in Prolog to denote a list. In the case of the <path list> used by the show() command, we have a list of lists. These data structures are the foundation of the modular Petri net model. Implementation of the validation tool was feasable because of Prologs excellent list processing ability.

<token> indicates that the place at a specified path contains a token with an integer value. (More precisely, <token> is stored at port(out) of the place). When <token> is used for pattern matching for a given state of the model, <wild card> says that any integer value is acceptable. Temporarily, the analysis program uses -99 as the wild card, on the assumption that protocol models use only positive integers.

```
<token list> ::=  []  |
                  [ <token> {,<token>} ]

<token> ::= token( <path> , <integer> )
          | token( <path> , <wild card> )
```

<path> ::= [] |

[<place name> {,<place name>}]

<path list> ::= [] |

[<path> {,<path>}]

<event list> ::= [] |

[<event> {,<event>}]

<event> ::= event(<path> , <transition name>)

11. Extension to Packet Petri net model.

The current model is a modular numerical Petri net. Tokens carry integer values, to enable the modelling of protocols. The computational power of the model could mushroom, if tokens were allowed to carry lists of integers, i.e. packets. This is a feasable extension to the current program, given the list processing features of Prolog.

The following extension would be required to the identification of a token.

<token> ::= token(<path> , <packet>)

<packet> ::= [] |

[<integer> { , <integer> }]

APPENDIX - G

"Needle" Specification of Alternating Bit Protocol

Contents:

The alternating bit protocol provides reliable message flow between sender and receiver via an unreliable channel. The model is specified in "Needle", a specification language for the modular numeric Petri net model. The "Needle" specification language is implemented on top of the Prolog language. Validiation is carried out by the Prolog program TRAV.PRO, which processes the specification G.1 and prints the event sequence G.4.

Listing: G.1 **(Filename: ALT.PRO)**

NEEDLE SPECIFICATION OF ALTERNATING BIT PROTOCOL

Places store sequence count as an integer:
Use the 'idle' place with values of 0 and 1 to represent two states 'idle0' and 'idle1'. Similarly the 'wait' place represents two states 'wait0' and 'wait1'. Otherwise the specification corresponds exactly to the schematics (Appendix-E).

```
predicates
  spec
clauses
  spec :-

  begin_spec,

  /*** Model execution definitions ***/

  initial([ token([sender,idle],0),
            token([receiver,wait],0) ]),
  end_option(state),
  end_state([ token([sender,idle],1) ]),


  /*** Model specification ***/


  /* Sender module. Transmits data packets from its output port. */

  module(s_mod),
  port(in), port(out),
    place(elementary,idle),       /* idle place stores integer 0,1 */
    place(elementary,wait),       /* wait place stores integer 0,1 */
      transition(send_d0), get(idle,0),
                           put(wait,0), put(out,0),
      transition(send_d1), get(idle,1),
                           put(wait,1), put(out,1),
      transition(s0_rx_err), get(wait,0), get(in,-1),
                             put(wait,0), put(out,0),
      transition(s1_rx_err), get(wait,1), get(in,-1),
                             put(wait,1), put(out,1),
      transition(s0_rx_a1), get(wait,0), get(in,1),
                            put(wait,0), put(out,0),
      transition(s1_rx_a0), get(wait,1), get(in,0),
                            put(wait,1), put(out,1),
      transition(s0_rx_a0), get(wait,0), get(in,0),
                            put(idle,1),
      transition(s1_rx_a1), get(wait,1), get(in,1),
                            put(idle,0),
  end(s_mod),
```

(continued...

Listing: G.1 (continued)

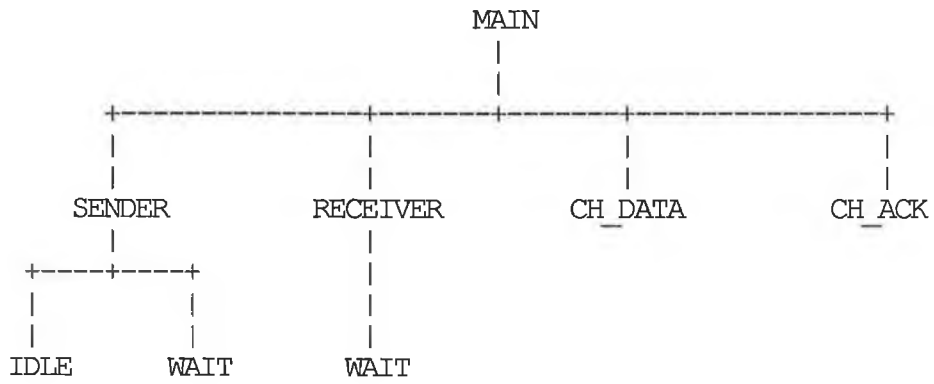
```
  /* Channel module.  Transmits, corrupts or loses messages. */

  module(c_mod),
  port(in), port(out),

        transition(tx_msg), from(in), to(out),
        transition(tx_err), from(in), put(out,-1),
        transition(lose_mg), from(in),

  end(c_mod),


  /* Receiver module. Inputs data packets and echoes ACK pacckets. */

  module(r_mod),
  port(in), port(out),
    place(elementary,wait),        /* wait place stores integer 0,1 */

     transition(r0_rx_err), get(wait,0),get(in,-1),
                            put(wait,0),put(out,1),

     transition(r1_rx_err), get(wait,1),get(in,-1),
                            put(wait,1),put(out,0),

     transition(r0_rx_d1), get(wait,0), get(in,1),
                           put(wait,0), put(out,1),

     transition(r1_rx_d0), get(wait,1), get(in,0),
                           put(wait,1), put(out,0),

     transition(r0_rx_d0), get(wait,0), get(in,0),
                           put(wait,1), put(out,0),

     transition(r1_rx_d1), get(wait,1), get(in,1),
                           put(wait,0), put(out,1),
  end(r_mod),


  /* Main module. Defines the communication paths between sender */
                    /*  and receiver via data and ack channels */
  module(main),
    place(s_mod,sender),
    place(r_mod,receiver),
    place(c_mod,ch_data),
    place(c_mod,ch_ack),
         transition(s_tx), from(sender),   to(ch_data),
         transition(r_rx), from(ch_data),  to(receiver),
         transition(r_tx), from(receiver), to(ch_ack),
         transition(s_rx), from(ch_ack),   to(sender),
  end(main),

end_spec.
```

Listing: G.2

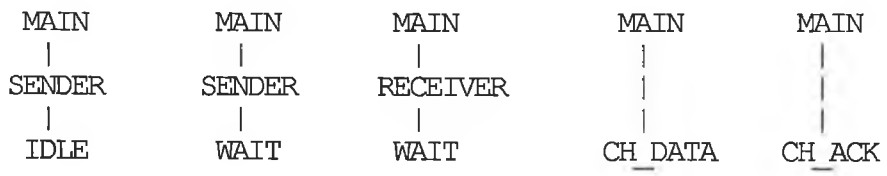
HIERARCHICAL STRUCTURE OF THE MODEL

The model system is designed as a hierarchy of modules. **MAIN** is the top level module. The **SENDER, RECEIVER** and two **CHANNEL** modules are at the next level. Elementary Petri net places are at the lowest level. The transitions are also in the hierarchy, though not represented here.

Each place in the hierarchy is identified by its path, specified as a Prolog list. For example, the empty list **[]**, is the path of the top level **MAIN** module. **[SENDER,WAIT]** is The path of a **WAIT** place in the SENDER module.



The following figure represents translation of the hierarchy into a flat Petri net consisting of individual places, all at the same level. Five of these places **[SENDER,IDLE]**, **[SENDER,WAIT]**, **[RECEIVER,WAIT]**, **[CH_DATA]** and **[CH_ACK]** are shown below:



Listing: G.3

Translation of modular specification.

The specification of modules and submodules is translated to a list of Petri net places and transitions.
Each place is uniquely identified by its path:
[module, submodule, submodule...]
The main module is identified by its path: [].
Transitions are identified by their path and event name.

```
SPECIFIED PLACES
  []
  ["sender"]
  ["sender","idle"]
  ["sender","wait"]
  ["receiver"]
  ["receiver","wait"]
  ["ch_data"]
  ["ch_ack"]
  ["deadlock"]  /* predefined place */


SPECIFIED EVENTS
  []s_tx                  /* main module */
  []r_rx
  []r_tx
  []s_rx
  ["sender"]send_d0         /* sender module */
  ["sender"]send_d1
  ["sender"]s0_rx_err
  ["sender"]s1_rx_err
  ["sender"]s0_rx_a1
  ["sender"]s1_rx_a0
  ["sender"]s0_rx_a0
  ["sender"]s1_rx_a1
  ["receiver"]r0_rx_err      /* receiver module */
  ["receiver"]r1_rx_err
  ["receiver"]r0_rx_d1
  ["receiver"]r1_rx_d0
  ["receiver"]r0_rx_d0
  ["receiver"]r1_rx_d1
  ["ch_data"]tx_msg      /* channel modules */
  ["ch_data"]tx_err
  ["ch_data"]lose_mg
  ["ch_ack"]tx_msg
  ["ch_ack"]tx_err
  ["ch_ack"]lose_mg
```

Listing: G.4

Full Event Sequence of Alternating Bit Protocol.

The test conditions specify that the sender module is initially in **IDLE** state with sequence count **'0'**, and must execute an event sequence which returns to that state. The event sequence is over two cycles of transmission of data and receipt of ack. Sequence count is **0** in the first cycle and **1** in the second cycle. No errors occur.

```
Initial state:                    /* Test conditions */
  ["sender","idle"] 0
  ["receiver","wait"] 0
Final conditions:
  ["sender","idle"] 0
Avoid events:
  1 ["ch_data"] lose_mg
  2 ["ch_data"] tx_err
  3 ["ch_ack"] lose_mg
  4 ["ch_ack"] tx_err


18 events to final state:          /* Test result */
  ["receiver","wait"] 0
  ["sender","idle"] 0

Valid event sequence:

  1 ["sender"] send_d0
  2 [] s_tx
  3 ["ch_data"] tx_msg            /* DATA 0 --> RECEIVER
  4 [] r_rx
  5 ["receiver"] r0_rx_d0
  6 [] r_tx
  7 ["ch_ack"] tx_msg             /* SENDER <-- ACK 0
  8 [] s_rx
  9 ["sender"] s0_rx_a0

  10 ["sender"] send_d1
  11 [] s_tx
  12 ["ch_data"] tx_msg           /* DATA 1 --> RECEIVER
  13 [] r_rx
  14 ["receiver"] r1_rx_d1
  15 [] r_tx
  16 ["ch_ack"] tx_msg            /* SENDER <-- ACK 1
  17 [] s_rx
  18 ["sender"] s1_rx_a1

  (Note: [] = main module )
```

APPENDIX - H

SPECIFICATION TRAVERSAL PROGRAM - TRAV.PRO

INTERACTIVE MENU SCREENS

Contents:

H.1 COMPILING THE TRAVERSAL PROGRAM

The Turbo Prolog programming environment is used on an IBM compatible personal computer. Two program source files are required:

TRAV.PRO - The specification traversal main program

ALT.PRO - The specification under analysis, in this case the alternating bit protocol.

Execute Turbo Prolog and continue interactively by use of its menus. Load TRAV.PRO and edit the last line:

```
include "alt.pro"
```

so that the specification under analysis will be included. Finally compile TRAV.PRO with the included file.

H.2 THE ASSIST MENU AND ANALYSIS MENU

Run TRAV.PRO, interactive operation is carried out in the dialog window of the Turbo Prolog environment. The prompt "goal:-" appears on the screen. Enter the assist command to bring up the main menu:

```
+----------------Prolog Dialog Window---------------+
| goal:-  assist <ret>                              |
+---------------------------------------------------+

+---------------TRAV.PRO    Assist Menu-------------+
|DIALOG - Return to Prolog goal: prompt             |
|Analysis Commands - Execute analysis of the model  |
|Terminal Conditions - Set up initial & final cond's|
|Event Control - Set up of valid event sequences    |
|Search Control - Set up properties of search algor'm|
|Display Control - Set up format of displayed results|
|Display Model - Display internal structure of model |
+---------------------------------------------------+
```

Select "Analysis" from the main menu. Analysis of the specification is in two phases initiated by the following commands:

DO spec - Translate the specification ALT.PRO into a flat Petri net model, store it in the Prolog database.

DO search - Search the execution paths of the model, and display the results.

```
+----------TRAV.PRO    Analysis Commands----------+
|RETURN                                           |
|DO spec - Model specif'n (Resets test conditions)|
|DO search - Search event sequences of model      |
+-------------------------------------------------+
```

Select "DO spec", processing the specification and bringing up the specified experimental conditions in the test status window.

```
+-----------------Test Control Status---------------+
|depth(30), search depth                            |
|Initial state:                                     |
|  ["sender","idle"] 0                              |
|  ["receiver","wait"] 0                            |
|end_option(cycle)                                  |
|Show nodes:                                        |
|  ["chan_data"]                                    |
|  ["chan_ack"]                                     |
|Option: tree(yes), reachability tree.              |
+---------------------------------------------------+
```

H.3 CHOOSE TERMINAL CONDITIONS

A terminal condition is the specified state of the model system at the beginning or end of its event sequence. Initial conditions specify the inital marking of the Petri net; the position of all tokens and the numeric value which they contain. The final marking specifies the required final position of all tokens. The search algorithm searches for every possible event sequence which will lead from the initial marking to the final marking.

Instead of terminating on final marking, it is possible to terminate the event sequence on deadlock situation, or on looping to a previous marking of the model.

The experimental test conditions are adjusted interactively via the available menus. For example the following two menus provide choice from three different "end_options".

```
+-----------TRAV.PRO    Terminal Conditions----------+
|RETURN                                              |
|initial(TOKENS) - Initial state of model system.    |
|end_option - Select option which ends event sequence|
|end_state(TOKENS) - Conditions for final state      |
+----------------------------------------------------+
```

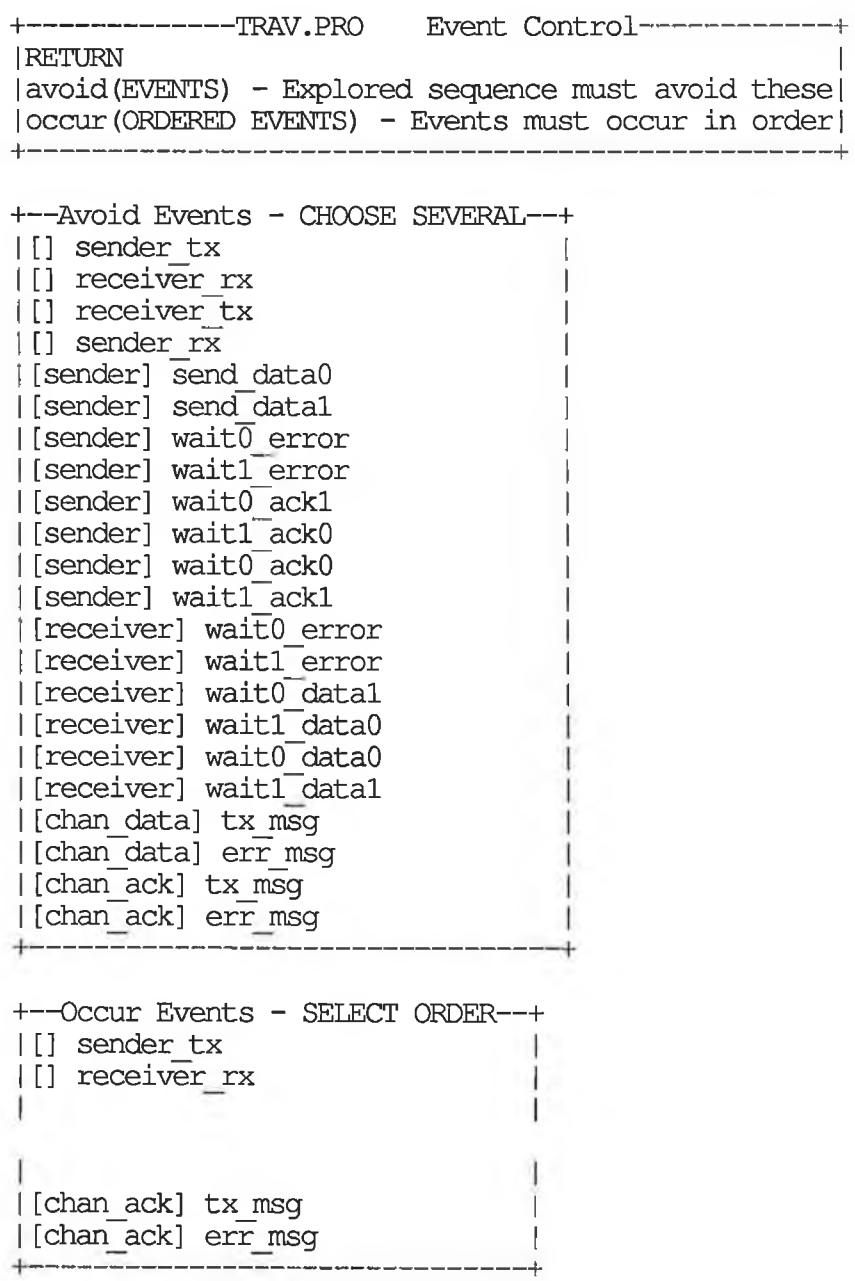
Selecting "end_option" brings up a menu of three alternative options:

```
+-----------------Select End Option-----------------+
|NO CHANGE                                          |
|cycle -  Search until model cycles to previous state|
|deadlock - Search until no further events can occur |
|state - Search until specified end state satisfied  |
+---------------------------------------------------+
```

H.4 EVENT CONTROL

The search algorithm identifies possible execution sequences of the specified model. By selecting events from two pick lists, it is possible to ask complicated questions such as:

Can events 'A' and 'C' occur in that order,
-----
avoiding event 'B' ?



H.5 CONTROL OF SEARCH ALGORITHM

Control is provided over the search algorithm. The first control, search depth specifies an integer value. The last two controls are YES/NO options.

```
+------------TRAV.PRO    Search Control------------+
|RETURN                                            |
|depth - Limit length of event sequence searched   |
|permit_loops - Enable looping by search algorithm |
|first_result - Enable search of one or all results|
+--------------------------------------------------+
```

H.6 CONTROL OF DISPLAY FILTER

Control is given on the amount of information displayed during an event sequence search. The first two options control a filter, displaying events at selected modules only.

The TREE option chooses between event sequence display or display of the reachability tree. The last option enables a debug display.

```
+-------------TRAV.PRO    Display Control------------+
|RETURN                                              |
|Show All - Enable events to be shown at all places  |
|show(PLACES) - Show events only at selected places  |
|Select Tree -  Sequence display or reachability tree|
|Select Track -  Debugs search algorithm             |
+----------------------------------------------------+
```

H.7 DISPLAY OF PETRI NET COMPONENTS

The "Display Model" menu displays the Petri net components of the specified model. The result of selecting "Modules" and "Places" is shown below:

```
+----------TRAV.PRO    Display Model----------+
|RETURN                                       |
|Modules - Display specified modules and ports|
|Places - Display all instances               |
|Transitions - Display all instances          |
+---------------------------------------------+
```

```
Modules:          Ports:

elementary        ["out"]          /* Basic place */
send_module       ["in","out"]
channel_module    ["in","out"]
receive_module    ["in","out"]
main              []
```

```
Places:
  []
  ["sender"]
  ["sender","idle"]
  ["sender","wait"]
  ["receiver"]
  ["receiver","wait"]
  ["chan_data"]
  ["chan_ack"]
  ["deadlock"]
```

H.8 SEARCH GENERATES REACHABILITY TREE

Having set up the test conditions, running the 'DO search' commmand from the 'Analysis' menu, will display a reachability tree as follows: The "- - -" symbol indicates "same as above", and the joints between the branches are omitted, as a join up the dots puzzle !
Their is one event sequence (from left to right) displayed per line.

```
+----------TRAV.PRO    Analysis Commands----------+
|RETURN                                           |
|DO spec - Model specif'n (Resets test conditions)|
|DO search - Search event sequences of model      |
+-------------------------------------------------+


Enter filename to store results,
or press RETURN for screen display:
depth(30), search depth
Initial state:
  ["sender","idle"] 0
  ["receiver","wait"] 0
end_option(cycle)
Show nodes:
  ["chan_data"]
  ["chan_ack"]
Option: tree(yes), reachability tree.

tx_msg  tx_msg  tx_msg  tx_msg
- - -  - - -  - - -  err_msg tx_msg
- - -  - - -  - - -  - - -   err_msg
- - -  - - -  err_msg tx_msg
- - -  - - -  - - -  err_msg
- - -  err_msg tx_msg
- - -  - - -  err_msg
err_msg tx_msg
- - -  err_msg

Search for all solutions complete -
9 valid sequences end in cycles.
Depth sufficent to find all cycles.
Press RETURN for Assist menu
```

APPENDIX - I

ANALYSIS OF PROTOCOL ERROR RECOVERY

Contents:

I.1 Recovery from error in data message.

Event sequence from transmission of data to receipt of ack. Force a data message error showing recovery. Listings are direct output from TRAV.PRO program.

```
Initial state:                    /* Test conditions */
  ["sender","idle"] 0
  ["receiver","wait"] 0
Final conditions:
  ["sender","idle"] 1
Occur event sequence:
  1 ["ch_data"] tx_err          /* Force a data error */
Avoid events:
  1 ["ch_data"] lose_mg
  2 ["ch_ack"] lose_mg

17 events to final state:         /* Test result */
   ["receiver","wait"] 1
   ["sender","idle"] 1
Valid event sequence:
  1 ["sender"] send_d0
  2 [] s_tx
  3 ["ch_data"] tx_err        /*  DATA 0 --> ERROR
  4 [] r_rx
  5 ["receiver"] r0_rx_err
  6 [] r_tx
  7 ["ch_ack"] tx_msg         /*  <-- ACK 1
  8 [] s_rx
  9 ["sender"] s0_rx_a1
  10 [] s_tx
  11 ["ch_data"] tx_msg       /*  DATA 0 -->
  12 [] r_rx
  13 ["receiver"] r0_rx_d0
  14 [] r_tx
  15 ["ch_ack"] tx_msg        /*  <-- ACK 0
  16 [] s_rx
  17 ["sender"] s0_rx_a0
```

Listing: I.2

Recovery from errors in data or ack messages.

Event sequence from transmission of data to receipt
of ack. Force errors on each channel showing recovery.
The 'show' command selects relevant events for display.

```
Initial state:                    /* Test conditions */
  ["sender","idle"] 0
  ["receiver","wait"] 0
Final conditions:
  ["sender","idle"] 1
Show nodes:                       /* Show events on
  ["ch_data"]                        the channels only */
  ["ch_ack"]
```

Force data message error

```
Occur event sequence:             /* Test conditions */
  1 ["ch_data"] tx_err

17 events to final state:         /* Test result */
  ["receiver","wait"] 1
  ["sender","idle"] 1

Valid event sequence:
  3 ["ch_data"] tx_err            DATA 0 --> ERROR
  7 ["ch_ack"] tx_msg             <-- ACK 1
  11 ["ch_data"] tx_msg           DATA 0 -->
  15 ["ch_ack"] tx_msg            <-- ACK 0
```

Force ack message error

```
Occur event sequence:             /* Test conditions */
  1 ["ch_ack"] tx_err

17 events to final state:         /* Test result */
  ["receiver","wait"] 1
  ["sender","idle"] 1

Valid event sequence:
  3 ["ch_data"] tx_msg            DATA 0 -->
  7 ["ch_ack"] tx_err             ERROR <-- ACK 0
  11 ["ch_data"] tx_msg           DATA 0 -->
  15 ["ch_ack"] tx_msg            <-- ACK 0
```

APPENDIX - J

PROTOCOL DEADLOCK RECOVERY BY TIMEOUT

Contents:

J.1 Specify Timer Module and Timeout Transitions

Define a timer module which is activated by deadlock of the system. Use this timer within the sender module to activate the s0_timeout or s1_timeout transitions. The rest of the specification is unchanged, only changes are listed.

```
/* Timer module */                                          /** NEW  **/
                                                            /** CODE **/
module(t_mod),                                              /**      **/
  port(out),                                                /**      **/
      transition(elapse), from(deadlock), to(out),          /**      **/
end(t_mod),                                                 /**      **/


/* Sender module */

module(s_mod),
port(in), port(out),
  place(elementary,idle),
  place(elementary,wait),
  place(t_mod,timer),                    /***  NEW TIMER MODULE ***/
    transition(send_d0), get(idle,0),
                         put(wait,0), put(out,0),
    transition(send_d1), get(idle,1),
                         put(wait,1), put(out,1),
    transition(s0_rx_err), get(wait,0), get(in,-1),
                           put(wait,0), put(out,0),
    transition(s1_rx_err), get(wait,1), get(in,-1),
                           put(wait,1), put(out,1),
    transition(s0_rx_a1), get(wait,0), get(in,1),
                          put(wait,0), put(out,0),
    transition(s1_rx_a0), get(wait,1), get(in,0),
                          put(wait,1), put(out,1),
    transition(s0_rx_a0), get(wait,0), get(in,0),
                          put(idle,1),
    transition(s1_rx_a1), get(wait,1), get(in,1),
                          put(idle,0),
                                                  /** NEW TRANSITION **/
    transition(s0_timeout), get(wait,0), from(timer),
                            put(wait,0), put(out,0),
                                                  /** NEW TRANSITION **/
    transition(s1_timeout), get(wait,1), from(timer),
                            put(wait,1), put(out,1),
end(s_mod),
```

Listing: J.2

Timer recovers from lost data message.

Event sequence from transmission of data to receipt of ack. Force a lost data message showing that timer **elapse** and **s0_timeout** provides recovery from deadlock.

```
Initial state:                  /* Test conditions */
  ["sender","idle"] 0
  ["receiver","wait"] 0
Final conditions:
  ["sender","idle"] 1
Occur event sequence:
  1 ["ch_data"] lose_mg  /* Force lost data message */
Avoid events:
  1 ["ch_data"] tx_err
  2 ["ch_ack"] tx_err

13 events to final state:       /* Test result */
  ["receiver","wait"] 1
  ["sender","idle"] 1

Valid event sequence:
  1 ["sender"] send_d0
  2 [] s_tx
  3 ["ch_data"] lose_mg         /*  DATA 0 --> LOST
  4 ["sender","timer"] elapse
  5 ["sender"] s0_timeout
  6 [] s_tx
  7 ["ch_data"] tx_msg          /*  DATA 0 -->
  8 [] r_rx
  9 ["receiver"] r0_rx_d0
  10 [] r_tx
  11 ["ch_ack"] tx_msg          /*  <-- ACK 0
  12 [] s_rx
  13 ["sender"] s0_rx_a0
```

Listing: J.3

Reachability Tree of Deadlock Sequences

```
This analysis explores all possible event sequences,
which lead the model to a terminal state.  Since the
model is of a cyclic protocol system, terminal states
represent the unwanted deadlock condition.
The results generate NO tree, therefore we have a
deadlock free protocol.


depth(30), search depth          /* Test conditions */
Initial state:
  ["sender","idle"] 0
  ["receiver","wait"] 0
end_option(deadlock)
Show nodes:
  ["ch_data"]
  ["ch_ack"]
Option: tree(yes), display reachability tree.
Option: permit_loops(no), search until loop.

/* Test result */

Search for all solutions complete -
No valid sequence to deadlock.
Depth sufficent to find all cycles.
```

REFERENCES

[AGHA86] Agha, G.
"Actors: A model of concurrent computation in distributed systems (TEXTBOOK)",
MIT Press 1986 , series in artificial intelligence,
ISBN 0-262-01092-5

[AJMO86] Ajmone Marsan, M.
"Performance models of multiprocessor systems (TEXTBOOK)",
MIT press series in computer science,
1986 ISBN 0-262-01093-3 , 280 pages

[ANAG86] Anagnostou, M. and Protonotarios, E.
"Performance analysis of the selective repeat ARQ protocol",
IEEE Trans. Communications,
vol. COM-34, No. 2, Feb. 1986, pp. 127-135

[ANDE85] Anderson, D. and Landweber, L.
"A Grammer Based Methodology for Protocol Specification and - Implementation",
Proc. 9th Data Comms Sympos. IEE Comput. Soc. Press,
vol. 1985 pp. 63-70

[ARCH87] Archetti, F.
"Petri net modeling of a highly concurrent machine",
CH2480-2/87/0000-0389 (C) 1987 IEEE,
vol. pp 389-393

[BART83] Barto, A. et al
"Neuronlike adaptive elements that can solve difficult learning control problems",
IEEE Trans. Systems, Man and Cybernetics,
vol. SMC-13, No. 5, Sept. 1983 pp. 834-846

[BERZ88] Berzins, L.
"Rapidly prototyping real time systems",
IEEE Software,
vol. Sept. 1988 pp 25-36

[BILL85] Billington, J. (see also pn8806)
"On Specifying Performance Aspects of Protocol Services",
International Workshop on Timed Petri Nets. Conference proceedings Turin Italy.,
vol. 13 July 1985, pp. 288-295

[BILL88] Billington. J., Wheeler, G. and Wilbur-ham, M.
"PROTEAN: A high level Petri net tool for the specification - and verification of communication protocols",
IEEE trans software eng,
vol. 14, No. 3, Mar 1988, pp. 301-316

APPENDIX - K References

[BOUC87] Boucher, C. and Bouvet, C.
"Designing a simulation language on the basis of composite Petri nets",
Proc. 1987 summer computer simulation conf. Publ. sandiego - CA VSA:SCS 1987, Montreal Que. canada,
vol. 27 Jul 1987, pp. 701-706

[BRIN83] Brinksma, E.
"An Algebraic Language for the Specification of the Temporal-Order of Events",
EUTECO European Teleinformatics Conference,
vol. 1983 pp. 533-542

[BRUN86] Bruno, G.
"Process translatable Petri nets for rapid prototyping of process control systems",
IEEE Trans. Software Eng.,
vol. Feb 1986 pp 346-357

[BURT84] Burton, F. and Huntbach, M.
"Virtual tree machines",
IEEE Trans Comput.,
vol. c-33, No. 3, Mar 1984, pp. 278-280

[BURT87] Burton, F.
"Functional programming for concurrent and distributed computing",
The Computer Journal,
vol. 30, No.5, 1987, pp. 437-

[BURT88] Burton, F.
"Storage management in virtual tree machines",
IEEE trans computers,
vol. 37, No. 3, Mar. 1988, pp. 321-328

[CAVA87] Cavalli, A. and Horn, F.
"Proof of specification properties by using finite state machines and temporal logic",
Protocol spec, testing and verification VII, Elsevier Science Pub.,
vol. VII, 1987, pp. 221-233

[CHAN89] Chang, C.
"Modeling a Real time multitasking system in a timed PQ net",
IEEE Software,
vol. Mar. 1989 pp 46-51

[CHAR83] Ansart, J., Chari, V., Simon, D. and Rafiq, O.
"VADILOC: A Protocol Validator and PDIL: A Language for Protocol Description and Implementation.",
EUTECO European Teleinformatics Conf. (Italy),
vol. 1983, pp. 543-556

APPENDIX - K References

[CHOI85] Choi, T.
"Formal Techniques for Specification, Verification and Construction of Protocols",
IEEE Communications Magazine,
vol. 23, No. 10, 1985, pp. 46-52

[CHOU87] Chou, C.
"Performance evaluation of concurrent programs modeled by timed PQ-Net",
Proc.Compasc 87 11th Inter.Conf.Software & Applic. Conf. 1987
Cat. 87CH2447-1 Tokyo 7th Oct pp. 465- (IEEE CompSoc press)

[CIAR87] Ciarlo, A.
"Satellite On-Board Applications of Expert Systems",
European Space Agency Journal,
vol. 11, No. 1, 1987, pp. 31-44

[CLAR86] Clarke, E., Emerson, E. and Sistla, E.
"Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications",
ACM Transactions on Programming Languages and Systems.,
vol. 8, No. 2, April 1986, pp. 244-263

[CLOC87] Clocksin, W. and Mellish, C.
"Programming in Prolog",
Springer Verlag publ.,
1987, 3rd ed., ISBN 0-378-17539-3, 281 pages

[COHE89] Cohen, G.
"Algebraic tools for Performance evaluation of discrete event systems",
Proc. IEEE,
vol. 77 No. 1, Jan 1989, pp 39-57

[DAGO85] D'Agostini, V. and Grillo, M.
"Experiences in OSI Protocol Specification Using SDL",
CSELT Tech.Rep.(Italy),
vol. XIII, No. 3, June 1985, pp. 167-173

[DUGA89] Bechta, J.
"Stochastic Petri net analysis of a replicated file system",
IEEE trans, Software eng.,
vol. 15, No. 4, Apr 1989 pp 394-401

[ELLO86] Elloy, J. and Roux, O.
"Electre: A Language for Control Structuring in Real Time",
The Computer Journal,
vol. 29, No. 3, 1986, pp. 229-234

APPENDIX - K References

[FAN_88] Fan, I.
"A Prolog simulator for interactive flexible manufacturing systems control",
Simulation (CA. USA),
vol. June 1988, p. 239-247

[FELD86] Feldbrugge, F. (editor Brauer,W.)
"Petri net tool overview 1986",
Petri Nets: Applications and Relationships to other models of concurrency,
LNCS vol 255, Berlin: Springer-Verlag, Feb 1987, pp 20-61

[FLEI87] Fleischmann, A., Chin, S. and Effelsberg, W.
"Specification and implementation of an ISO session layer",
IBM Systems Journal,
vol. 26, No. 3, 1987, pp. 255-275

[FRIE89] Frieder, O.
"Protocol verification using database technology",
IEEE jour. Selected areas of communications,
vol. 7, No. 3 April 1989, pp 324-334

[GALL87] Gallard, R.
"An extension in the definition of a Petri net execution",
Comput. Jour.,
vol. 30, No. 1, 1987, pp. 16-19

[GARG85] Garg K.
"An Approach to Performance Specification of Communication Protocols Using Timed Petri Nets",
IEEE Transactions on Software Engineering,,
vol. SE-11, No. 10, Oct. 1985, pp.1216-1225

[GERS89] Gershwin, S.
"Hierarchical Flow control: A framework for scheduling and planning discrete events in manufacturing systems",
Proc. IEEE,
vol. 77, No. 1 Jan 1989 pp 195-

[GIAC88] Giacalone, A.
"Integrated Environments for formally well founded design and simulation of concurrent systems",
IEEE Trans. Software Eng.,
vol. 14 No. 6, June 1988, p. 787-801

[HARE88] Harel, D.
"On visual formalisms",
Communications of the ACM,
vol. 31, no. 5, May 1988, pp. 514-530

[HASP89] Hass, P.
"Stochastic Petri net Repres'n of discrete event simulations",
IEEE trans Software Eng.,
vol. 15, No. 4 Apr. 1989 pp 381-393

APPENDIX - K References

[HASS89] Hassard, John.
"Light at End of the tunnel. (Fundamental Particle Physics).",
New Scientist (U.K.),
vol. 8th July 1989, pp. 65-67

[HOAR78] Hoare, C.
"Communicating Sequential Processes",
Commun. ACM,
vol. 21, 8 (Aug 1978) pp 666-677

[HOLL87] Holliday, M. and Vernon, M.
"Exact Performance Estimates for Multiprocessor Memory and Bus Interface.",
IEEE Trans. Computers,
vol. C-36 no. 1, Jan 1987, pp. 76-85

[HOLZ87] Holzmann, G.
"On limits and possibilities of automated protocol analysis",
Protocol specification, testing and verification proc. IFIP-wg 6.1, 7th Internat conf. Zurich Switzerland,
vol. 5th May 1987, North Holland Amst., pp. 339-344

[HO 89] Ho, Y.
"Proc. IEEE. Special Issue on dynamics of discrete event systems",
Proc. IEEE,
vol. 77 No. 1 Table of contents, Jan 1989

[INAN89] Inan, K.
"Algebras of discrete event models",
Proc. IEEE,
vol. 77, No. 1, Jan 1989 pp 24-38

[JARD88] Jard, C. et al
"Development of Veda, a prototyping tool for distributed algorithms",
IEEE Trans software eng.,
vol. 14, No. 3, mar 1988, pp. 399-

[JENS75] Jensen and Wirth
"Pascal User Manual and Report",
1975.

[JENS87] Jensen, K.
"Coloured Petri nets",
CH2480-2/87/0000-0395 (C) 1987 IEEE,
pp 395-401

[JONS83] Jonsson, B. and Pehrson, B.
"An Extended Finite State Machine Approach to Specification, Validation and Testing of Protocols.",
EUTECO European Teleinformatics Conference,
1983 pp. 569-576

APPENDIX - K References

[KARA88] Karam, G.
"An icon based design method for Prolog",
IEEE Software,
vol. July 1988 pp 51-65

[KAVI86] Kavi, K., Buckles, B. and Bhat, U.
"A Formal Definition of Data Flow Graph Models",
IEEE Transactions on Computers,
vol. C-35, No. 11, Nov. 1986, pp. 940-947

[KIM_87] Kim, D., Choi, H. and Choi, Y.
"An automated protocol validation tool based on extended numerical Petri net.",
Tencon 1987, Seoul, S.Korea,
IEEE 1987, ch2423-2/87/0000-0351

[KOES67] Koestler, A.
"The ghost in the machine",
Picador, Pan Books Ltd. U.K.,
ISBN 0 330 24446 9 , year 1976, written 1967

[KOSK87] Kosko, B.
"Constructing an associative memory",
BYTE (USA),
vol. Sept 1987 pp. 137-144

[KOUN88] Kountanis, D.
"Petri net representation of rule based expert systems",
Western Michigan University, Kalamazoo, Mich 49008 USA,
1988, p143-152

[KRIT86] Kritzinger, P.
"A Performance Model of the OSI Communication Architecture",
IEEE Trans. Communications.,
vol. COM-34, No. 6, June 1986, pp. 554-563

[LAIR87] Laird, J. et al.
"SOAR: An architecture for general intelligence",
Artificial Intelligence,
vol. 33, 1987 pp. 1-64

[LEBL83] LeBlanc, T, and Cook, R.
"An analysis of language models for high performance communications in Local Area Networks",
ACM 0-89791-108-3/83/006/0065,
1983, pp. 65-72

[LEE_85] Lee, K. and Favrel, J.
"Hierarchical Reduction Method for Analysis and Decomposition of Petri Nets",
IEEE Transactions on Systems, Man and Cybernetics,
vol. SMC-15, No. 2, March 1985, pp. 272-280

APPENDIX - K References

[LEE_88] Lee, T. and Lai, M.
"A relational algebraic approach to protocol verification",
IEEE Trans Software Eng,
vol. 14, No. 2, Feb 88, pp. 184-

[LIN_86] Lin, T. and Mead, C.
"A Hierarchical Timing Simulation Model.",
IEEE Transactions on CAD,
vol. CAD-5, No. 1, Jan. 1986, pp. 188-197

[LOON88] Looney, C.
"Fuzzy Petri nets for rule based decision making",
IEEE Trans. Systems, Man and Cybernetics,
vol. 18, No. 1, Jan. 1988, pp. 178-183

[MARS87] Marsan, A., Chiola, G., Fumagalli, A.
"Timed Petri net model for the accurate performance analysis of CSMA/CD bus LANs",
Computer Communications,
vol. 10 No. 6, December 1987, pp. 304-312

[MART83] Martikainen, O. and Ahtianen, A.
"Modular Specification of OSI Subsystems",
EUTECO European Teleinformatics Conf. (Italy),
vol. 1983, pp. 587-597

[MART86] Martin, J.
"Abstract Data Networks and Application to Communication Protocols. IN FRENCH - Reseaux de Donnees Abstrait...
Thesis of L'universite Paul Sabatier de Toulouse (Sciences),
No. 3327, Jul. 1986, pp. 1-164

[MCAL87] McAllister, C.
"The Simulation of Systems as Timed Sequences of Events",
National Institute for Higher Education, Dublin,
Working paper CA-0387, Oct 1987, pp. 1-61

[MEAN88] Meandzija, B.
"Archetype: A unified method for the design and implementation of protocol architectures",
IEEE Trans. software eng.,
vol. 14 No. 6, june 1988, p. 822-837

[MIER87] Mier, S. and Talavage, J.
"A hybrid paradigm for modeling of complex systems",
Simulation (USA),
vol. 48, No. 4, Apr 1987, pp. 135-141

[MILN85] Milne, G.
"CIRCAL and the representation of communication, concurrency and time",
ACM Trans. Prog. Lang. and Systems,
vol. 7, No. 2, Apr. 1987, pp. 270-298

APPENDIX - K References

[MOLL89] Molloy, M.
"Special section on Petri net Performance models",
IEEE trans. Software Eng.,
vol. 15, No. 4, Apr 1989

[MOTO85] Moto-oka, T. and Kitsuregawa, M.
"The fifth generation computer",
Wiley 1985,
ISBN 0 471 90739 1

[MURA88] Murata, t. and Zhang, D.
"A predicate-transition net model for parallel interpretation of logic programs",
IEEE trans software eng,
vol. 14, No. 4, apr 1988, pp. 481-497

[NIXO89] Wing, J. and NIXON, M.
"Extending Ina Jo with Temporal Logic",
IEEE trans. software eng.,
vol. 15, No. 2 feb 1989 pp 181-197

[PACH87] Pachl, J.
"Protocol description and analysis based on a state transition model withchannel expressions",
Protocol specification, testing and verification VII Zurich-Switzerland, North Holland Amsterdam,,
vol. 5 May 1987, pp. 207-219

[PETE89] Peterka, G.
"Proof procedure and answer extraction in Petri net model of logic programs",
IEEE trans. Software eng.,
vol. 15, No. 2, Feb 1989 pp 209-217

[PETR62] Petri, C.
"Kommunikation mit automaten",
Schriften des Rheinisch-Westfalischen Institute fur Instrumentelle Mathematic an der Universitat Bonn,
W. Germany 1962.

[PETR66] Petri, C.
"Communication with automata",
Rome Air Development Center, Griffiths Air base, New York, USA,
vol. RADC-TR-65-337, Jan 1966

[PS8901] Sidhu, D.
"Formal methods for protocol testing: A detailed study",
IEEE Trans. Software Eng.,
vol. 15, No. 4, Apr 1989

APPENDIX - K References

[RAMA85] Ramamoorthy, C., Dong, S. and Usada, Y.
"An Implementation of an Automated Protocol Synthesiser (APS) and its Application to the X.21 Protocol",
IEEE trans. Software Eng.,
vol. SE-11, No. 9, Sept. 1985 pp. 886-908

[REED88] Reed, J. and Yeh, R.
"Specification and verification of liveness properties of cyclic concurrent processes",
ACM Transactions on Programming Languages and Systems,
vol. 10, No. 1, Jan. 1988, pp. 156-177

[REGG88] Reggiani, M. and Marchetti, F.
"A proposed method for representing hierarchies",
IEEE Trans. Systems, Man and Cybernetics,
vol. 18, No. 1, Jan. 1988, pp. 2-8

[ROUX85] Roux, J.
"Analysis of Distributed Systems using Timed Petri Nets. IN FRENCH - Analyse des Systemes Distributes ...",
Thesis of L'Institut National des Sciences Appliquees de Toulouse,
vol. No. 156, Dec. 1985, pp. 1-149

[ROZE87] Rozenberg, G.
"Advances in Petri nets 1987 (with bibliography of 2000 papers)"
Pub. Springer Verlag (Berlin),
vol. Apr. 1987, pp. 309-451

[RUDI83] Rudin, H.
"From Formal Protocol Specification Towards Automated Performance Prediction.",
Protocol Specification, Testing and Verification III, IBM Zurich,
vol. III, May 1983, pp. 257-269

[SANT88] Santos, A., Freitas, V. and Neves, J.
"The specification and prototyping of communication protocols from heterogeneous temporal logic to concurrent prolog",
Ibercom 87 1st Iberian conf on data communications vol 2,
Computer Communications System, Elsevier Science Pub. 1988

[SARI85] Sarikaya, B. and Bochmann, G.
"A Test Design Methodology for Protocol Testing",
Proc. 18th Hawaii Int. Conf. on System Sciences,
vol. 2, 1985, pp. 710-721

[SCHI88] Schindler, M.
"Computer-supported graphics ease software specifications.",
Electronic Design,
vol. 3 Mar. 1988, pp. 87-98

APPENDIX - K References

[SEIT84] Seitz, C.
"Concurrent VLSI architectures",
IEEE Trans computers,
vol. C-33, No. 12, Dec 1984, pp. 1247-1265

[SEVI88] Sevinc,S & Zeigler, B.
"Entity structure based design method: LAN protocol example"
IEEE Trans. Software Eng.
vol. 14, No. 3, Mar 1988.

[SHAF87] Shafer, D.
"Advanced Turbo Prolog programming (TEXTBOOK)",
Howard W. Sams & Company 1987,
ISBN 0-672-22573-5

[SHAT87] Shatz, S.
"Petri net modeling and analysis of the LAPD protocol standard"
Proc.Compasc 87 11th Inter.Conf.Software & Applic. Conf. 1987,
Cat. 87CH2447-1 Tokyo 7th Oct pp. 694-700 (IEEE CompSoc press)

[SIDH86] Sidhu, D. and Blumer, T.
"Verification of NBS Class 4 Transport Protocol",
IEEE Transactions on Communications,
vol. COM-34 No. 8, August 1986, pp. 781-789

[SIDH88] Sidhu , D. and Crall, C.
"Executable logic specifications for protocol service interfaces",
IEEE trans software eng,
vol. 14, No.1, Jan 1988, pp. 98-121

[TABA85] Tabak, D. and Levis, H.
"Petri Net Representation of Decision Models",
IEEE Trans. Systems, Man and Cybernetics.,
vol. SMC-15, No. 6, Nov. 1985, pp. 812-818

[UCHI87] Uchihira, N.
"Concurrent program synthesis with reusable components using temporal logic",
Proc.Compasc 87 11th Inter.Conf.Software & Applic. Conf. 1987,
Cat. 87CH2447-1 Tokyo 7th Oct pp. 455-64, (IEEE CompSoc press)

[VALK78] Valk, R.
"Self-Modifying nets, a Natural extension of petri Nets",
Automata, Languages and Programming, Udine (Lecture notes vol. 62),
Springer-Verlag (Berlin) 1978 pp. 464-476

[VERN87] Holliday, M. and Vernon, M.
"A generalised timed Petri net model for performance analysis",
IEEE trans software engineering,
vol. SE-13, no. 12, Dec 1987, pp. 1297-1310

APPENDIX - K References

[VUON88] Vuong, S. , Lau, A. and Chan, R.
"Semiautomatic implementation of protocols using an Estelle-C compiler",
IEEE trans software eng.,
vol. 14, No. 3, Mar 1988, pp. 384-

[ZEIG87] Zeigler, B.
"Hierarchical modular discrete event modelling in an object oriented environment",
Simulation (USA),
vol. 49, No. 5, Nov 1987, pp. 219-230

[ZOBR87] Zobrist, G.
"Modified computational graph and its usage in concurrent design",
CH2480-2/87/0000-0385 (C) 1987 IEEE,
pp 385-388