INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

TECHNIQUES FOR SPECIFICATION AND VALIDATION OF COMPLEX PROTOCOLS

THEODORE J. EWANCHYNA

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

DECEMBER 1996

Abstract

Techniques for Specification and Validation of Complex Protocols

Theodore J. Ewanchyna

Simple protocols can be described with natural language and pictures. This description is no longer adequate as protocols become more complex, or more people get involved. Formal Description Techniques (FDTs) have been developed to convey this information clearly and unambiguously (and as a basis for correctness verification) but FDT-based specifications tend to be long and hard to follow.

In order to help the user understand and use complex protocols, we have integrated informal graphical techniques with the current formal techniques. We have developed an approach for the generation of pictures directly from formal models.

Acknowledgments

First, I would like to thank my supervisor for his intellectual and financial support.
I would especially like to thank Gustave Arroyo for his help in organizing and editing portions of this thesis.
I wish to acknowledge and thank the support staff in both the Computer Science and the Electrical and Computer Engineering departments of Concordia University and also the Computer Science staff at the Université de Montréal.
My thanks also to Dr. Holzmann, the creator of the PROMELA language and the SPIN tools, for the frequent correspondence over the years via e-mail.
Finally, I would like to acknowledge and thank my father, Wesley Ewanchyna for his help and support during the final and crucial phase of the writing of this thesis.

To the memory of Frank Thomson

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Computer protocols were first examined as an interesting problem in computer science with what became known as the "alternating bit protocol" [Lyn68]. A simple English and graphic description delineated the rules for two computers to follow in order to reliably transfer packets of information between them. But even this simple protocol description was found to be in error [BSW69] and, ominously, serves as a warning to other protocol designers: designing and describing protocols is deceptively complex—it looks easy to describe the mechanisms that make up what we call a protocol, but it is not! As modern protocols achieve levels of complexity many times the level of complexity of 'toy protocols' like ABP, the need for a more precise formulation becomes obvious.

1.1.1 FDTs

Formal methods or formal description techniques (FDTs) are used to unambiguously describe how protocols work. The designer can use these techniques throughout the entire protocol design life cycle, from the initial design to the final validation and testing stages. As well, the protocol user (or implementer) can employ the formal description as the authorative definition of how the protocol works.

When employing FDTs, an unambiguous model of the protocol system is created using a formal language. Most formal languages were created to support the first design stage for protocols, the specification stage. An important aspect of using FDTs is that once a protocol

design is completed, it can be formally verified and validated. FDTs allow one to unambiguously describe protocols; but that is not enough, we must offer some assurance that the protocol is correct.

> *Unfortunately, when the design of a new protocol is complete, we usually have little trouble convincing* ourselves *that it is trivially correct. It can be unreasonably hard to prove those facts formally and to convince also others.* [Hol93]

With formal techniques, a protocol description can be verified to be correct before it is implemented, that is, before it is put out into the field and used. Furthermore, most protocol descriptions are machine readable so that these descriptions can be shown automatically to be correct.

1.1.2 Protocol Visualisation

Prior to using FDTs, protocol designers used natural language descriptions and pictures to explain how protocols work. Several factors led to designers adopting formal methods:

- as designers examined their 'success' with 'informal methods', they noticed inaccuracies and errors
- modern protocols became more complex
- designers' abilities to describe modern protocols using 'informal methods' could no longer keep up.

Despite the advantages of using FDTs, however, many protocol designers still do not use them. The few protocol designers that use FDTs do not use them throughout the *entire* protocol design life cycle. In fact, we can see two camps, the 'formalists' and the 'informalists'.

Some disadvantages of using FDTs are that training people to use them takes some extra time and FDT-based specifications tend to be large and hard to follow.

On the other hand, informal methods are intuitive so that they are relatively easy for human beings to use, as we are very good at recognising patterns and gleaning a great deal of information from illustrations.

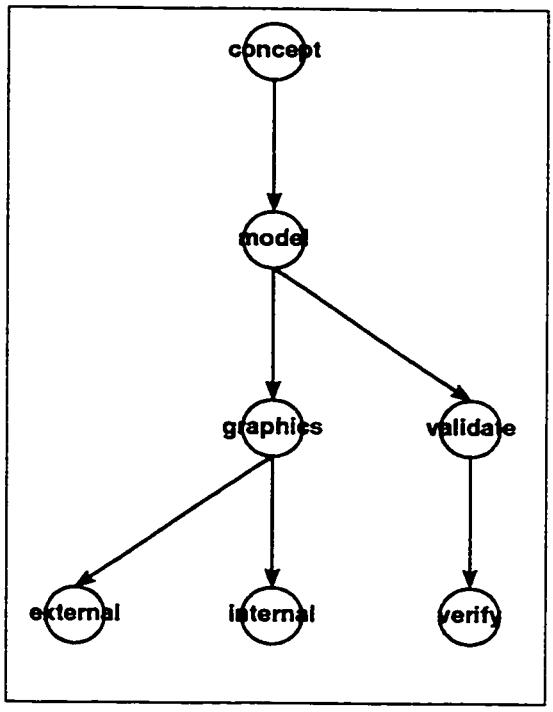


Figure 1: Design Methodology Overview

If formal methods are so superior to informal methods, why are not more people using them? As such, it may benefit us to re-examine what we hope to accomplish by building a formal model. We find that FDTs are most often used to:

- find problems in an existing informal specification
- prove the merit of FDTs by showing how easy it is to use FDTs
- as a basis for a proof step
- as a basis for other phases of protocol design steps.

We believe that we would be better off asking who is the specification (or model) being written for—humans or computers? The answer is for 'both'. An informal model is useful for humans and a formal model is potentially useful to humans but definitely must be 'formal' if it is to be used by computers.

We feel it would be beneficial to combine the two approaches: we propose using FDTs to create formal models; these can be validated and verified as is done already with FDTs

and can also be used as a basis for generating graphics of the system.

Figure 1 represents our design methodology . We see that we first start off with a concept; next, we express this concept as a model, preferably formally. Then we can validate and then verify the model. We can also produce graphics from it, wherein the graphics represent the model 'internally' and 'externally'[1].

Our design methodology allows both formal and informal description techniques. If we follow the 'right' path after the 'model' node, we represent the employment of standard FDTs. If we follow the 'left' path after 'model', the 'graphics' node represents the use of graphics from either an informal or formal model, so that this path also illustrates the ability to include an informal design approach independent of FDTs or in unison with their use.

New [New91, NA91, NA89a, NA89b] has done much work in the area of graphical representation of protocols, or what he calls "protocol visualization". He asserts that protocol design should incorporate graphical representation because this is 'natural' for us.

> *The most relevant portions of human cognitive and visual processing are the ability to automatically recognise proximity and connectivity between pictorial elements, the ability to automatically recognise inclusion of some pictorial elements within others, and the ability to automatically focus attention on changing elements of the visual field; all of these traits were encouraged by human evolution.* [New91]

This author also argues that a three pronged approach that uses the respective strengths of computers, humans, and FDTs (or as he terms it 'specification') should be used. Figure 2 shows this approach graphically.

1.1.3 Complex Protocols

The Xpress Transport protocol (XTP) is one complex protocol that was created specifically to address the demands of a new generation of networking and distributed computing needs that older protocols could no longer meet. Previous protocols, such as those in the OSI and TCP/IP protocol suites, were designed for environments running on Ethernet or telephone dialup lines, and their main applications were remote logins and file transfers.

[1]This idea will be elaborated on in the next chapter.

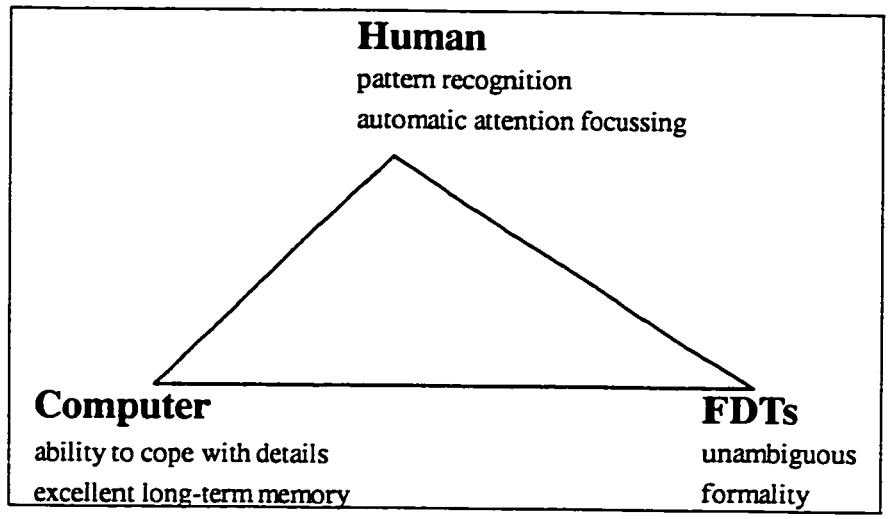


Figure 2: Protocol Visualisation

With the advent of optical fiber technology and ATM networks, many new and previously unforeseen applications are now feasible and many old applications can be improved or even extended.

It follows that new protocols such as XTP would have to be developed in order to take advantage of new technologies. XTP was designed with these new technologies in mind. It has been optimised for the needs of the present day.

XTP is a large and complicated protocol and currently there is no complete, formal specification for it. The officially published description *informally* describes the protocol using natural language and pictures. Some implementation notes are provided but no formal description or analysis is available.

Much of the XTP definition concerns itself with describing 'mechanism' rather than 'policy'. As a protocol definition, this is expected but, without any background context it is hard for new XTP users to know how to use the protocol or for experienced XTP users to agree on how to use XTP's functionality. There is no 'service' description.

Many aspects of the XTP protocol are not very well defined – informally or formally, in either of the two base XTP documents: the XTP 3.6 definition document [Sil92] or the published text on XTP [SDW92]. We feel this situation to be unacceptable. Chesson, the inventor of XTP, in an accompanying letter to his protocol description, referring to the "imprecision in the protocol specification" writes:

The quantity of explanatory text is barely adequate, but the inclusion of an API[2] *interface may be a first step toward portability of applications across XTP implementations.* [Sil92]

and he looks forward to future drafts of the protocol, when this situation will be rectified:

Future revisions of XTP may include formal specification methods, such as state tables and language descriptions. ... An API fashioned in this manner may provide the long awaited service definition for XTP – a "bottom up" approach to service specification. [Sil92]

1.2 Scope of the Thesis

In this paper we investigate methods to describe complex protocols so that any ambiguities or imprecision can start to be cleared up in their descriptions. We apply the techniques of formal analysis to describe and validate our models in order to provide assurances that the model is both precise and correct. We also generate graphical descriptions as an aid to describe and understand the protocol. We use PROMELA as the protocol design and validation language.

The beginning chapters provides a look at the area of protocol engineering in general and then focusses on the use of the language PROMELA as a design tool for the specification, verification and validation of protocols. We look at finite state machines for specification, and at reachability analysis and temporal logic for validation and verification, respectively, then we compare the various techniques for formal methods. Although there are many techniques described in this chapter, only one, PROMELA, will be used to build protocol models.

Chapter 2 provides a description of what a protocol is. We use a definition by Holzmann to arrive at what he refers to as a 'complete' protocol description. The ISO OSI model is used to explain how a complex communications sytem is broken down into protocol description layers and the well defined interfaces that allow communication with those layers.

Chapter 3 provides a look at the techniques that have been developed to try and specify protocols. We start off describing informal desription techniques. Within this context we

[2]API is an acronym for "application-protocol interface"

provide a classification system based on 'internal' and 'external' descriptions of protocol systems. After this we introduce the topic of formal techniques.

As with all large software systems there exists a software life cycle. Protocol engineering is an attempt to provide a more sound protocol design technique much as software engineering does for non-communicating software systems. We discuss some of the specification, validation and verification techniques that have been used to try and come to terms with the complex issues of defining protocol systems.

After this chapter, we discuss the language, PROMELA, which we feel incorporates the best aspects of the techniques of formal protocol design. The language is not only used in the first phase, the specification phase but also has allowances for validation, verification and even simulation. Thus PROMELA can be used for all the phases of the protocol design life cycle, except for the final, implementation phase.

Next, we discuss aspects of graphics support. Chapter 5 reiterates the advantages of using graphics in protocol design, but this time in the context of a formal design approach. We briefly look at some examples that have influenced us and then take a look at some tools we have created in order to bring graphics tools into the PROMELA world.

Chapter 6 looks at the graphical tool we designed to draw out time sequence diagrams. We provide background and motivation for wanting to come up with such a tool and then describe how the tool works. Time sequence diagrams provide an 'external' description of the modules of the protocol system.

In chapter 7 we provide a similiar discussion for the graphical tool we designed to draw finite state machines. Finite state machines provide an 'internal' description of the modules of a protocol system.

The last chapter concludes the thesis and briefly examines other areas to explore.

Chapter 2

Protocol Definition

A protocol can be described as "the set of rules that two entities use in order to communicate with each other to accomplish a particular task". In order to show that the definition applies to many other things — other than computer 'protocols', this description is purposely as general as possible.

For example, when people engage in a telephone conversation they also follow a protocol: each person starts the connection with 'hello', talks for a while, and then says 'goodbye'. This, of course, abstracts the mechanics involved of dialling the telephone, taking the phone 'off hook', etc. See [BL93, BL91, SL93, FLS90a, FLS90b, FLS91], for several formal models of the telephone 'protocol' written in the formal specification language, LOTOS[1].

2.1 Computer Protocols

In the world of data communications, computers communicate with one another by transmitting messages between themselves. In order to assure reliable communications some sort of acknowledgement for the sender from the receiver—'error control' and other mechanisms are introduced. The sum total of the different mechanisms used are what defines a 'protocol'.

[1]The earlier papers discuss 'POTS' – plain old telephone system, while later papers build on this work and describe new telephone features like call waiting, etc. The problem of feature interaction, where newly introduced features interfere with already present ones, and different specification styles are mentioned when talking about these new features.

Equally important is for the protocol to be stated as a precise and unambiguous set of rules. Machines are not as flexible as humans. For example, where a human might 'hang up the phone' if no one is at the other end of the line a computer would not—unless it can recognize this situation and is specifically informed as to what to do. In other words computers may be fast but they are not all that smart. This then becomes the problem for protocol designers to overcome: how can we write a complete and unambiguous description so that computers can reliably transmit information between them.

2.1.1 Five Essential Ingredients

In an attempt to describe how to go about this, Holzmann [Hol91] enumerates five essential ingredients for a complete protocol specification:

1. the *service* to be provided by the protocol
2. the *assumptions* about the protocol in which the protocol is to be used
3. the *vocabulary* of messages used to implement the protocol
4. the *encoding* (format) of each message in the vocabulary
5. the *procedure rules* guarding the consistency of message exchanges

The *service* refers to the outside interface that is presented to the protocol user, which includes the available ports and messages that can flow through those ports. The *assumptions* can be regarded as a requirement to specify the 'extraneous' entities of the protocol model. This would include the environment that the protocol is to be run in and the types of users the protocol would expect to interact with. The *vocabulary* concerns itself with the correct sequence of messages that may pass through interconnecting ports. *Encoding (format)* refers to the exact sequence of information that is contained in a message. This can vary, depending on the level of abstraction that the model represents. Often, only a portion of the 'actual' message is defined. The designer is concerned only with the fields that are relevant to the properties being tested in the model, so that unused fields need not be entered. The *procedure rules* are the definition of the protocol itself. This includes how each defined module in a system behaves—its full set of rules that explain how to act in each and every situation for each and every module.

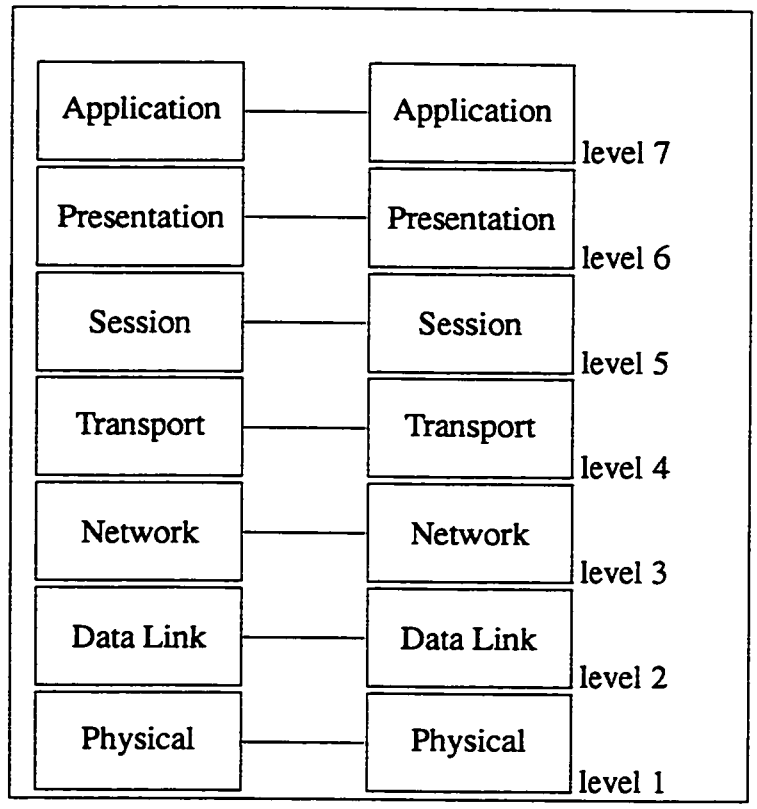


Figure 3: OSI Layers

What this means is that a protocol consists not only of the 'protocol rules' but equally important: the service that that protocol provides, the assumptions or environment that the protocol operates in, and the specification of the format and order (vocabulary) that is correct for a proper transmission to take place. Finally, these messages come from 'above' and from 'below' the protocol module or layer.

2.2 OSI Layers

Network architecture design has driven the design of communications systems. The methodology is to divide the functionality needed for system interconnection into separate modules. The layers are divided up in terms of a particular functionality. Each layer provides services to the upper layer and requests services from the lower layer using interactions on well defined boundaries via a small number of 'service primitives'. Dividing a system up like this

allows the user to form different abstractions at each level, which helps to break up the task of describing such a large system into manageable tasks. In this way each layer defines a protocol that defines its interaction with its remote peer module; with this (layer and interaction) description being the 'heart' of the protocol.

To give historical and terminological perspective to the reader, we first look at the International Organisation for Standardisation's (ISO's) OSI model [DZ83, Hea93]. The OSI BRM [ISO7498] introduced the concepts of layering and abstraction into the computer communications world in order to tame the complexity inherent in communications systems. The current standard "OSI Basic Reference Model", (see figure 3), breaks up a system into layers [Lin83, Liu89].

Each layer has a well defined service interface, where the environment is defined via the interface directly above and below it (except for the lowest (physical) layer which has no 'lower' layer and the topmost (application) which has no 'higher' layer). As part of this definition, the format and ordering of messages that pass through these interfaces is also defined.

To illustrate this approach, refer again to figure 3. The 'physical' layer forms the basis of the model. The 'data link' layer uses the services as expressed by the behaviour characteristics of the physical layer to create a service that the next higher (network) layer can use. How the data link layer does this is what we call its protocol description.

Starting from the data link layer, each higher layer adds a little more functionality to the lowest layer of 'bare metal' wires. Each layer uses the 'services' of the layer immediately below it to provide a service that the next higher layer can use which.

This idea is expressed more generally in figure 4 [Lin83, Liu89] which shows how a layer N protocol uses the services of layer N-1 to offer a service to the layer N+1, or the 'user'. Figure 5 [Lin83] shows this same model from the viewpoint of the user layer N+1) where the layer N protocol is seen to be a 'black box'. The concern of the user is 'service', not how that service is provided. How the service is provided is what we are concerned with defining, in other words, it *is* the protocol.

So for any given layer, the lower layer becomes part of the environment that the current layer can use. The current layer's protocol is designed to work with the lower layer to provide a service that the next higher layer can then use. This process repeats itself until the highest layer is defined.

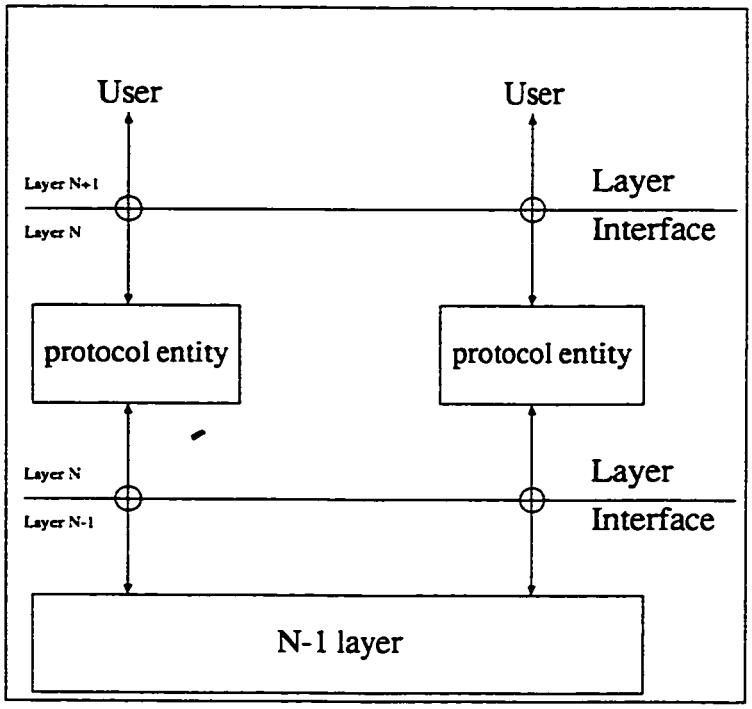


Figure 4: The Protocol Model

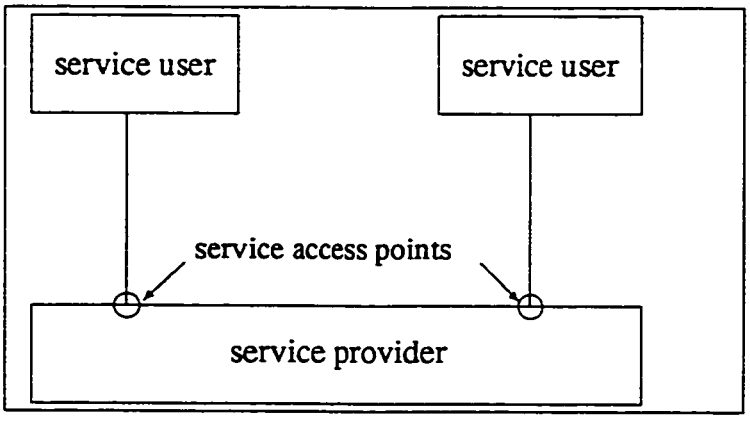


Figure 5: The Service Model

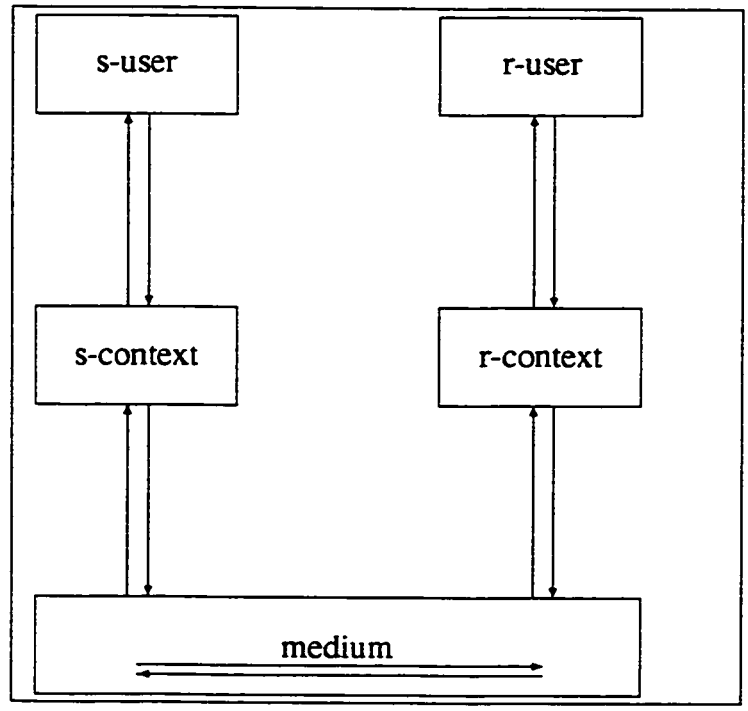


Figure 6: The "classic" architecture model

Figure 4 is often represented as shown in figure 6 when one wishes to describe the model's 'system architecture'. It should be clear how these last three figures relate to Holzmann's "five essential ingredients" of protocol specification as given in section 2.1.1.

2.3 Summary

It is a difficult job to define protocols clearly and correctly. Many techniques have been proposed in order to break the problem down into simpler sub-tasks. Abstraction and modularisation are used to break down the description of protocols into manageable units. The description techniques used to describe and design protocols are continually being refined as new approaches are discovered.

Chapter 3

Protocol Development Techniques

3.1 Introduction and History

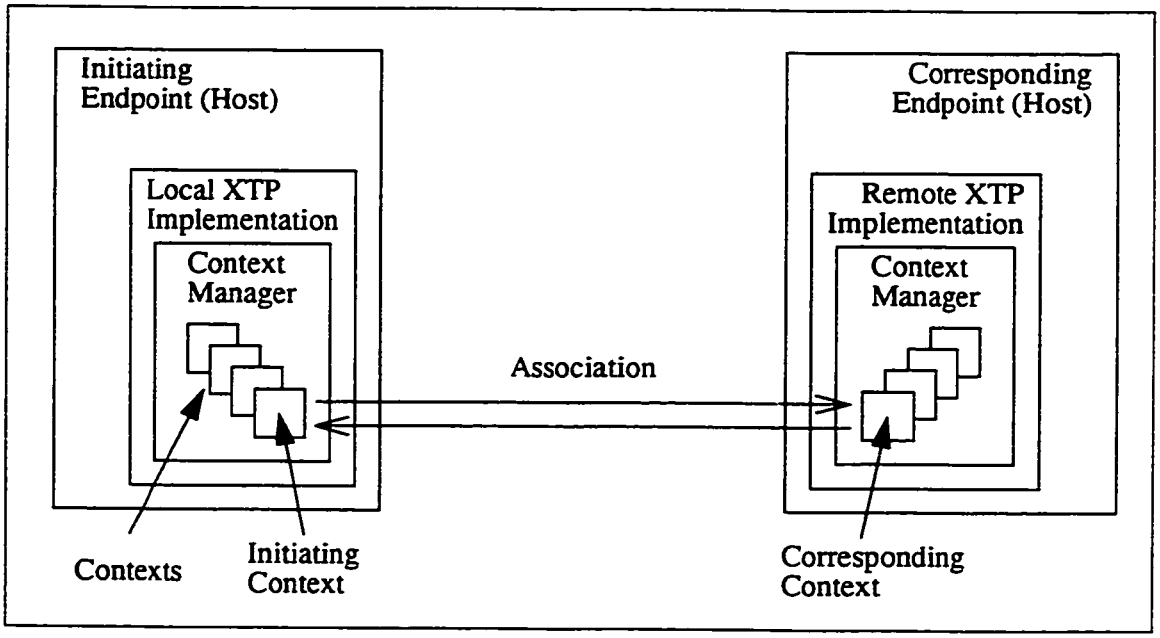
The application of FDTs to protocols was spear-headed by the International Organisation for Standardisation (ISO) in the 1980's in order to insure a high degree of telecommunications software dependability and quality (see [Vis90] for an introduction to the subject). Prior to FDT usage, only natural language descriptions, diagrams and code walk-throughs were used to describe protocols, but this did not suffice to specify the exact requirements for large software systems, such as protocol software systems.

3.2 Informal Methods

Protocols may quickly acquire a high degree of complexity, and informal descriptions fail to reflect this complexity, and may lead to mistakes in their implementation. On the other hand, formal descriptions are less likely to cause misunderstandings, and they may be automatically verified with the help of a computer.

For example, informal protocol design methods rely on the intuition and experience of the designer. Considering the complexity of protocol design this is a great compliment on the designer's abilities. However, not everyone has such great talent and even very experienced people can make mistakes [1].

[1] Imagine writing complex Pascal or C code by hand without recourse to a compiler and expecting the code to be right the first time through. Now imagine writing a complex system with several threads of control, *without* a methodology to follow and *without* a compiler. This is essentially how protocol designers worked



[XTP95]

Figure 7: Informal Architecture of XTP Communication Model

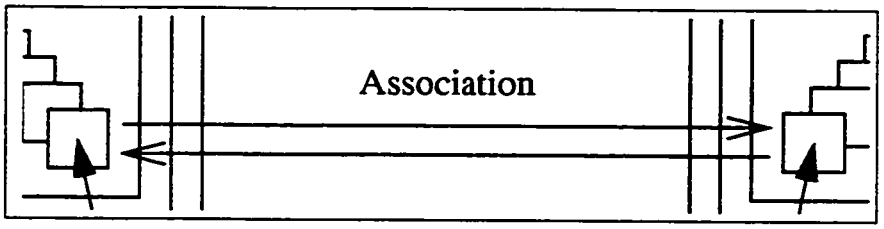
3.2.1 Describing a Protocol Informally

How would one go about describing a protocol informally? In the first place, the designer should try and convey a general overview of the protocol model. The participating system components and their inter-relationships should be described to show the modules and how they communicate with each other. Figure 7 shows an example of this overview for the XTP protocol architecture.

From this figure we see how a unicasting XTP 'association' can be built between two contexts (an initiating and its corresponding (listening) context and their corresponding two simplex data communications streams between them. Figure 8 is a closeup of this figure which we use to shift the reader's attention to this relationship. Each context exists on a different host which is running its own XTP protocol stack. As part of this implementation a 'context manager' is used to mediate the shunting of information to the proper context.

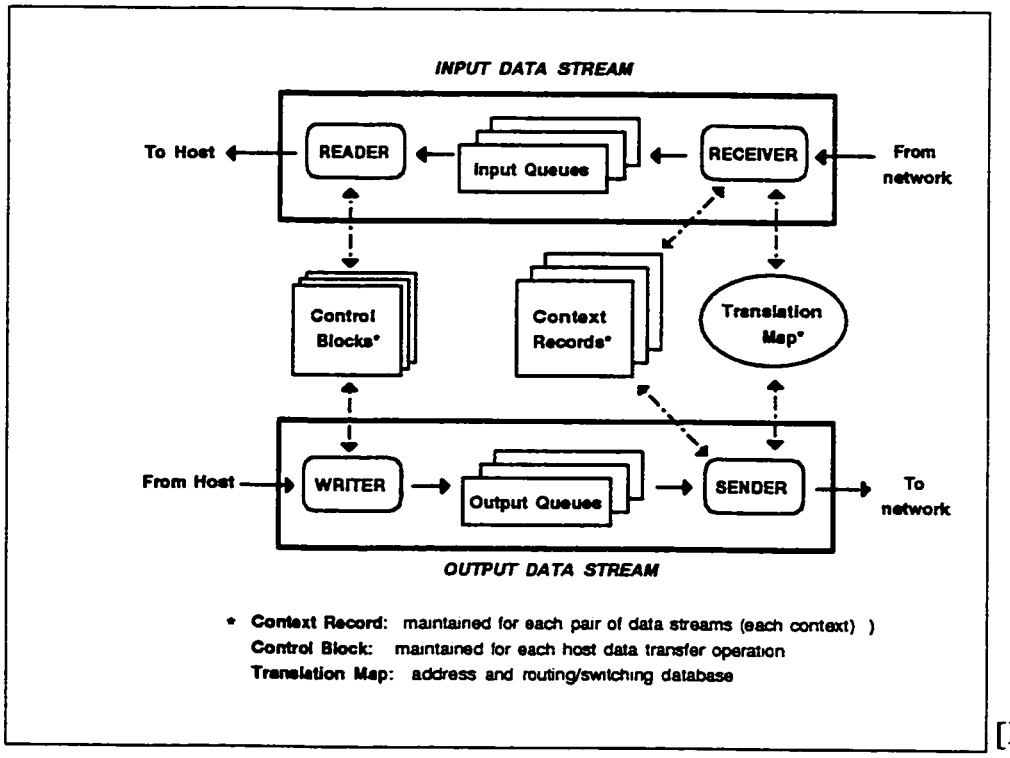
More detail and a better understanding is often provided by focussing on particular architectural components (see figures 7, 8, 9, and 10) and the designers of XTP have done just

before formal methods were adopted.



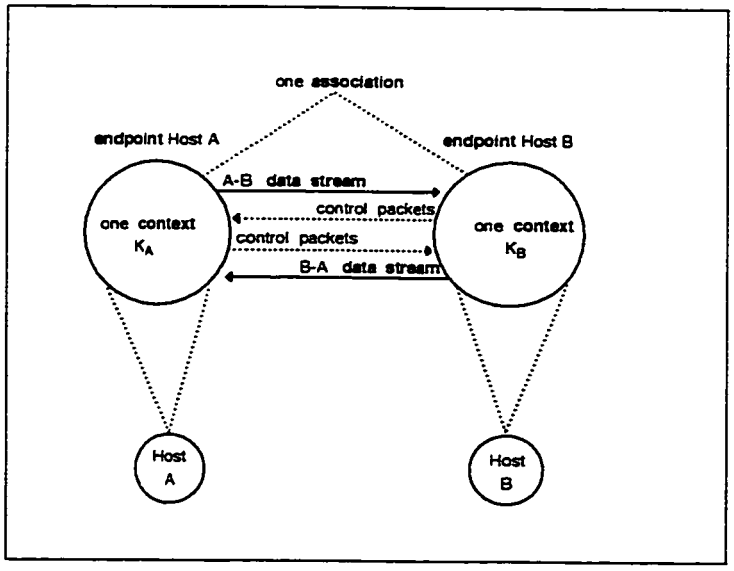
[XTP95]

Figure 8: Isolation of Contexts of the XTP Communication Model



[XTP95]

Figure 9: Internal Architecture of an XTP context



[Sil92]

Figure 10: XTP Architecture of Contexts with data streams

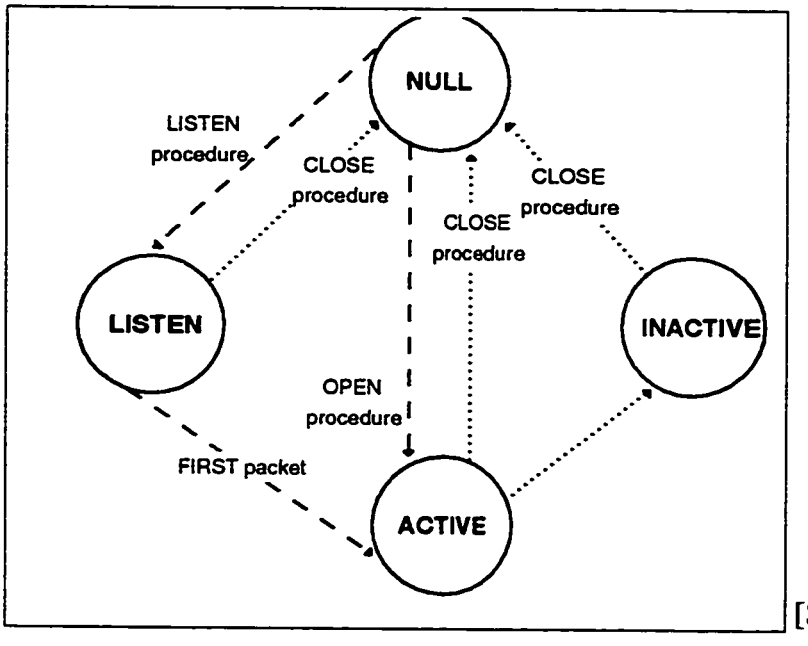
that in order to help describe their protocol. Figures 7 and 8 informally show an 'association' between two 'contexts' and how they are connected together (we produced figure 9). Figure 9 shows the makeup of a single XTP context. Figure 10 is similar to figure 8, but by focussing on the channels, it attempts to illustrate the communications between two XTP contexts; it shows how the contexts communicate with each other and the types of packets they exchange.

In this way a protocol model is gradually fleshed out by a series of increasingly detailed descriptions and illustrations. However, we note that the architecture only provides a static description. Once the various details of the architecture are provided, it is the events that occur and their consequences to the protocol system that often give a more informative description of the protocol. This allows a dynamic view of the protocol and these cause and effect events (or actions) can be described as occurring 'internally' and 'externally'[2].

Internal Description

As we break down the protocol into modules and processes that implement specific functionalities we find that at certain times a process is ready to either instigate (or 'offer') or respond to certain events and therefore *certain things can only happen at certain times.* For

[2]My terminology.



[Sil92]

Figure 11: XTP 3.6 Context state machine

example, a module may send a message and then wait for a reply. There may be several possible scenarios that may occur after the message is sent, and all of them must be accounted for.

Internal descriptions show how the modules are affected on a per process level. Using an example where a process has sent a message and is awaiting a response: the process progresses to a wait state and will only change its "state" upon receipt of the returned message. A different message may force the process into a different "state" or if no message arrives, this, too may force a change of state.

At each stage in a process' lifetime, a list of responses that are acceptable to it and how the module responds to them and how it changes (by going through a series of stages) is described. Assuming only legal responses occur, for example, the sending module only enters a new stage upon receipt of one of those responses. On the global scale, the entire protocol is described as the sum of all its component modules' executions and interactions. The protocol goes through each of its component modules and enumerates their expected events.

These "stages" are termed "states" in finite state machine terminlogy.

Figure 11 illustrates the general stages an XTP context goes through in its lifetime. The

reader should note that the actions listed on the transition arrows show the result of an outside process' communications with it, so that the process chooses to go from the "null' state to either the "listen" or the "active" state, as a result of information not shown in the figure. Informal descriptions are often "incomplete".

It must be pointed out that the transition arrows are not like flow chart arrows. For example, the transition from the "listen" to the "active" state indicates the passive reception of a "$FIRST$" packet (not the sending of one), as would be depicted in a flow chart. State transitions occur because of a process instigating or responding to an action.

In figure 11 every context starts off in a "null" or "quiescent" state. The context has two choices: it can open as an initiator, in which case it proceeds directly to the "active" state, or it can open as a responder, in which case it goes to the "listen" state and only upon receiving an appropriate "$FIRST$" packet does the context then proceed to the "active" state. Once a context is in an "active" state, it proceeds to the "inactive" state (generally when a data transmission session is completed).

Every state except for "null" has a "CLOSE procedure" transition that leads back to the "null"' state. When taken in the "inactive" state this transition represents a normal, uninterrupted cycle through the context's stages. Anywhere else, it represents an interruption in the normal data transmission session, usually due to some sort of unrecoverable error. Thus the finite state machine shows all possible behaviour *without* distinguishing between desired and undesired behaviours.

External Description

An "external" informal explanation shows the interactions between different modules that transmit messages to each other. *Message sequence diagrams* or *time sequence diagrams* [LL92b, LL92a, GR92c] are often used to explain the sequence of message exchanges between entities. These diagrams are used in most protocol descriptions. Directed arrows are drawn from the sender to the receiver of a given message. The module bodies are traditionally drawn along the x-axis and time is drawn along the y-axis starting from the top of the page (see figure 12).

There is an attempt to formalise this notation and turn it into a fourth FDT called **Message Sequence Charts** or MSCs. Grabowski and Rudolph [GR92c] gives a good introduction to the MSC language, including its history, modifications and enhancements, and

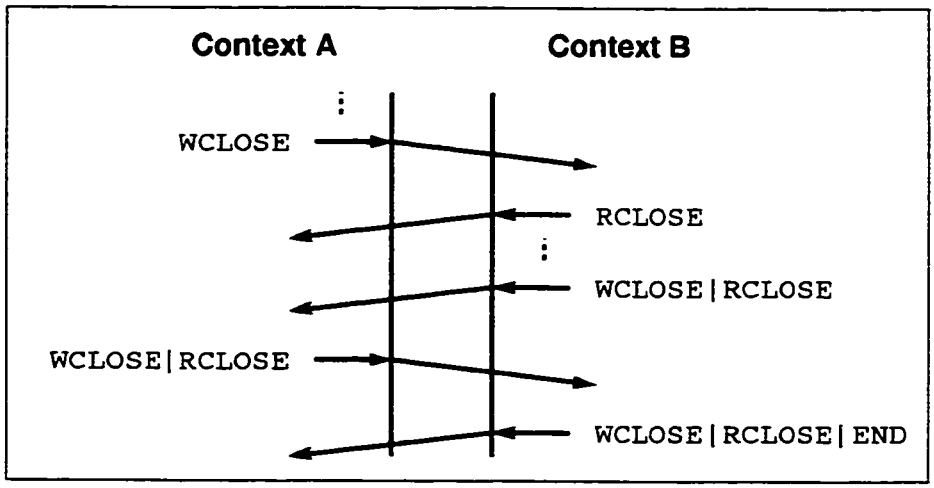


Figure 12: Fully Graceful Independent Close

the realisation that a clear semantics is not yet arrived at. An attempt to develop formal semantics for MSCs is given by Ladkin and Leue [LL92b], wherein the authors describe their work in turning MSCs into 'global state transition graphs' (GSTG) via "ne/sig graphs". They find that the MSC model, as defined, is under-specified because more than one GSTG can be drawn from a MSC specification. By looking at temporal logic and Büchi automata, however, they can give a formal semantics to MSC, by adding end states to the GSTGs that reflect the reliability properties of the communication (which is precisely what is under-specified by MSC notation) and correlate them with the accepted traces of the automata they wish to define [LL92a].

Sometimes a hybrid description approach is used. Figure 13 shows an example of the XTP protocol using a combination of an architecture diagram and a message sequence diagram in order to further convey some meaning about the protocol. This figure shows that a $FIRST$ packet must be the first packet sent, how it creates an association, and how this must precede all $DATA$ packet transmission.

Taken together the internal and external descriptions can give the user a good feel for how a protocol works and helps one to understand what types of problems the protocol solves and how it goes about solving them.

Often the specification is used by various people in different phases or stages of the protocol design life cycle: specification, verification, testing, implementation. Thus it must be

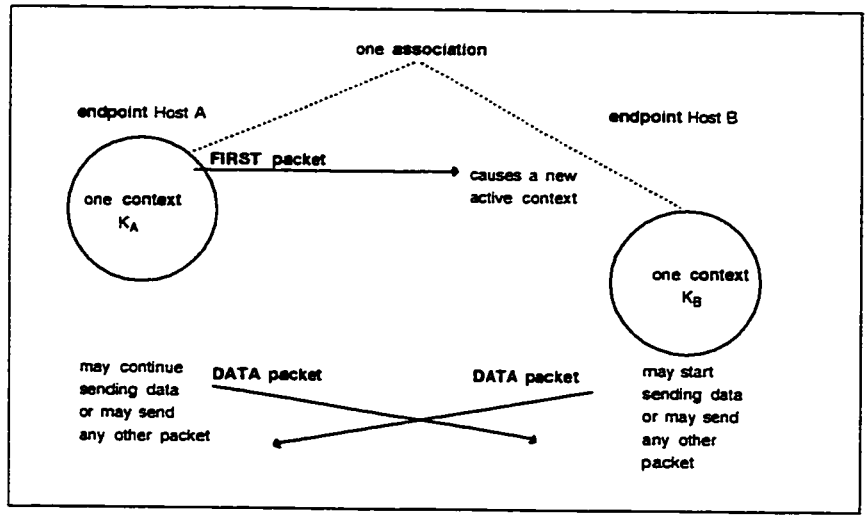


Figure 13: XTP in action: $FIRST$ and $DATA$ packet exchanges

clear, unambiguous, and sufficiently detailed. Only then can the specification give confidence that the protocol will be correct [Lin85]. As many potential interactions are unexpected and undesirable, FDTs help to spot problems in the specification early enough in the life cycle to ward off having to make expensive and difficult changes later on.

Without standard design techniques or standard formulations to specify and verify protocol designs, many problems arise. For example, each designer's techniques, style, and quirks have to be learned. Instead of making a 'black art' of protocol design (much as compiler design was in the 1960s [AU77]), we would like to develop or follow a methodology that others may actually use to correctly design complicated communications protocols.

Eventually, the notion of a formal description of a protocol is proposed. This technique is superior to informal methods because formal descriptions are precise to the point that now the code can be checked by machine [Mil90, Vis90, Liu89]. It is this approach which we feel holds the greatest promise and which we explore and build on in this thesis.

3.3 Formal Methods and Protocol Engineering

Protocol Engineering is the term used to describe the formal design methodology used throughout the entire protocol design life cycle—from the initial specification, to validation and testing, test case generation, and the generation of implementation code.

In order to accurately describe protocols, much work has been done in the field of formal description techniques (FDT). Most research has focussed on formally specifying, verifying and validating protocols (the first stages in protocol engineering) for logical correctness.

There is good reason for focussing on the earliest stages. Errors in the design stage have a nasty habit of showing up in the implementation stage, and when discovered, they are costly to fix.

3.3.1 Benefits of Formal Methods

FDTs provide a mathematical underpinning to specifications which ensures their precision and tractability. They allow for the design of open systems that insure implementations that can inter-work. The specifications are also less ambiguous and are free from implementation bias. FDTs attempt to provide a precise statement of the requirements.

Through their use, FDTs are found to [Tur93]:

- provide a high level of dependability and quality
- specify exactly what is required
- discover errors, ambiguities, and inconsistencies
- impose clear structure on the problem domain
- help in the early stages, to define the requirements and program structure
- highlight deficiencies in the existing informal description.

FDTs can also support the use of computer tools that can help the designer[3]. Now the tedious task of looking for inconsistencies can be handled by a machine and the designer can

[3]"It is not necessary to have computer tools to support FDTs since they can be used as intellectual tools." [Tur93]. This represents the strictly formalist point of view, wherein it is argued that model building and proof making should be done without any computer assistance. (For an extreme representation of this point of view see [Ste], where Dijkstra proposes teaching courses with programming languages which are not implemented on campus in order to discourage the use of computers in programming and to drive the point home that computer science is essentially a branch of formal mathematics.) We would reject this claim and say that computer tools are necessary; see Colin H. West in his foreword to [Hol91], where he says "Protocols have proved to be extremely difficult to understand without automated analysis tools".

focus on the more creative elements. Another advantage is that the 'bugs' can be eliminated at the earliest possible stage—before implementation coding has started.

In a major project, where several people are usually working together, descriptions must be formal and unambiguous. This means that the formal specification guides the design, not the interpretation of a diagram or natural language text fragment. It is important that all agree on one specification and the earlier they can do this the better.

3.3.2 Models

Models are often used to map from the human intuitive understanding of an application as expressed in natural language concepts to one based on more formal concepts. The term model is used in two senses (which should be clear from the context that it is used in). Specific specification languages are based on a conceptual model—process algebra or the finite state machine, for example. On the other hand, we build specific protocol system models in a particular specification language. Obviously this protocol system model will reflect, in the larger sense, whatever fundamental constructs are inherent in the specific specification language model. Models are chosen to represent the significant system properties that are required.

> *The purpose of a model is to precisely define specific properties or characteristics of a system under consideration to be built or analysed, and provides the foundation for verifying those properties ... thus ... one must select from a class of models which represent that property* [Lam85]

Models should allow all valid implementations but they often represent a tradeoff between good analytical powers and good expressive power. In the remaining section, we look at some models which we find useful for describing protocols.

For example, all three ISO FDTs use the "labelled transition system" (LTS) concept which is defined (rather generally) as "systems whose transitions between states are labelled with associated actions" as a basic model [Tur93]. We use this model (finite state machines) as well as another model which delineates the difference between concurrent and sequential programs. The first subsections look at the beginning of the research into this area.

Note: The following two subsections make frequent reference to the language PROMELA, which will be discussed in the next chapter. PROMELA relies heavily on the concepts

which follow and, as such we felt that it would be appropriate to contrast the original concepts with how PROMELA interpretes them *as we introduce the concepts.*

3.3.3 Guarded Commands

In the mid-1970's, Dijkstra [Dij75] applied some new structuring techniques he considered suitable to describe a certain class of sequential algorithms. He noted that these algorithms all shared the common trait of 'concurrency'. In having rewritten these algorithms, he proposed using 'guarded commands' to support the programming concept of 'non-determinism'. He noted that using non-determinism to express these algorithms made them easier to understand as their inherent concurrency was brought out.

Definition

The heart of the new structuring method is the guarded command. Structurally, a guarded command consists of two parts: a guard and a list of one or more commands.

```
guard -> command-list
```

Syntax

```
(x >= y) -> m := x
```

Example

Semantics

The guard indicates a boolean condition which controls the executability of an attached list of commands. When the guard is passed (if the boolean condition is true), the command(s) following the guard are executed. In the example above, if $x >= y$ then the command list (in this case, there is only one element in the list), $m := x$ is executed.

Guarded Command Sets

A single guarded command by itself does not differentiate this model from a sequential model of course. This paradigm is only useful when several guarded commands are grouped together to form a guarded command "set"[4]. With a guarded command set, the order that 'true' guarded commands in the set occur is not related to its choice of executability. So any

[4] "set" as used in the mathematical sense

guarded command that evaluates to true may be executed; sequential order, as for mathematical sets, is irrelevant. When $x == y$, for example, both $x >= y$ and $x <= y$ are equally possible, regardless of their order in the set. This is discussed more fully further on.

```
   guard0 -> command-list0
[] guard1 -> command-list1
```

Syntax

```
   x >= y -> m := x
[] x <= y -> m := y
```

Example

Two Language Structuring Constructs

Using the guarded command set as a primitive language construct, two language structuring methods are used:

- the alternative construct: $if < guardedcommandset > fi$, and
- the repetitive construct: $do < guardedcommandset > od$.

The Alternative Construct In an alternative construct, such as

```
   if
   x > y -> m := x
[] x < y -> m := y
   fi
```

at least one guard must be true or the program will abort. In the above example, program abortion occurs if x and y have the same value. The consequence of this, of course, is that the model can never block.

The semantics of this construct are different in PROMELA: the model will block if all the guards evaluate to false, i.e., no guard is true. Thus an 'exit' condition must be explicit (see left hand side of the following figure). In the latest version of PROMELA, this condition can be expressed more simply with the *else* statement (see right hand side of the following figure):

```
example:
        if
        :: x > y -> m := x
        :: x < y -> m := y
        :: x == y -> skip
        fi
```

PROMELA example

```
example:
        if
        :: x > y -> m := x
        :: x < y -> m := y
        :: else  -> skip
        fi
```

PROMELA version 2 example

Finally, as we have just seen, if more than one guard is true, a guarded command is chosen at random (or non-deterministically).

```
   if
   x >= y -> m := x
[] x <= y -> m := y
   fi
```

In the example, if x equals y both guards are true therefore either guard is non-deterministically chosen.

The alternative construct offers choices among alternatives, much like the CASE statement in Pascal or the SWITCH statement in C. It differs from C or Pascal, where, the first or most specific true choice is picked. Instead *any* 'true' or executable choice is randomly picked from the guarded command set.

The Repetitive Construct In a repetitive construct, a structure such as:

```
   do
   x > y -> x = x - y
[] y > x -> y = y - x
   od
```

is used for looping. When all guards are false the loop exits (for PROMELA, again, an explicit 'exit' condition is required or else the model blocks). After a guarded command finishes execution, control is returned to the beginning of the loop, and the choice of the next guarded command to be executed is repeated. This continues until an exit condition occurs.

Applying the Constructs

Concurrent algorithms are non-deterministic [5], thus repeated execution of the **same** algorithm, will result in a different sequence of execution for each subsequent run whereas execution traces of sequential programs are always the same on subsequent runs.

Proving Correctness and Algorithm Tracing Sequential programs are routinely proven correct the same way as mathematical functions are proven correct. For example, using a given domain and range, any input from the domain will always give the same result in the range. The result should be consistent and the steps executed to obtain a given output should also be consistent. Sequential algorithms are in essence 'functions'.

Any algorithm's execution trace can be followed by noting the order of steps taken every time the algorithm is run. For a sequential algorithm this trace must always be the same. This is not true for a concurrent algorithm.

Non-determinism The effects of non-determinism are most clearly seen after examining the execution traces of algorithms using constructs based on guarded commands. They show that the computation sequence and even the final state are not necessarily uniquely determined from the inputs.

For example, given two values x and y, one can not say which statement will execute if x and y are equal, as in:

```
example:
        do
        x >= 0 -> x = x - 1
      □ x >= 0 -> x = x - 2
        od
```

Re-writing Certain Classes of Algorithms In looking at how to write or 'derive' programs in this new style one of the algorithms Dijkstra considered was the GCD or Euclid's algorithm.

The following example calculates the GCD or greatest common divisor of any two numbers a and b. To illustrate, we can pick any numbers, so 23 and 47 will suffice. Note that this

[5] Other models do not equate concurrency with non-determinism. see [MP92]

example is essentially the same example that was used to introduce the repetitive construct. The result will always be the same (the greatest common divisor of a and b), but the order of execution or trace will not necessarily ever be the same for repeated runs. In other words, we do not know the order that these statements will execute.

```
x = 23; y= 47
        do
        x > y -> x = x - y
        y > x -> y = y - x
        od
```

(using guarded commands)

The next two examples are different implementations of the same algorithm but written without guarded commands. Notice that *two* implementations for the same algorithm can be written. Also note that neither version is as clear and concise as the version above.

```
x = 23; y= 47
while x <> y do
if x > y
 then x := x - y
 else y := y - x
fi
od
```

Version A

```
x = 23; y= 47
while x <> y do
while x > y do
 x := x - y od
while y > x do
 do y := y - x od
od
```

Version B

Dijkstra noted that several sequential versions exist to implement this algorithm, but each version has extraneous implementation details. Expressed using guarded commands, the algorithm is reduced to its bare essentials, hence making it easier to understand.

Concurrency

Dijkstra speculated on the possibility of concurrent execution of some of his algorithms. For example, when more than one guard is true, in a single CPU machine, only one alternative in a "do" loop may be executed at a time. However, in a multi-CPU environment, each alternative could potentially be executed by a separate CPU.

3.3.4 Communicating Sequential Processes

Hoare used Dijkstra's basic language structuring methods of alternation and repetition constructs (based on guarded commands) less in terms of non-deterministic algorithm design but, to further explore concurrency.

For example, single CPU computers are designed with the assumption that programs will run with a single thread of control: the model based upon a deterministic execution of a single sequential program.

With multi-CPU computer platforms, this is no longer true and so it is possible to rewrite certain algorithms to take advantage of potential concurrency by running several threads of control (parrallelism) in the program. This is not to say that a single CPU can't run this new algorithm: several processes do run in a time sharing environment 'virtually' at the 'same time' (the process scheduler uses small time slice periods to run each process so that each user has the impression that their process is the only one running), but it is more forthright when systems are explicitly designed to allow several threads of control to operate at once, such as in multi-CPU systems, thereby bringing out the real issues of concurrency.

Process Concept

This brings up new issues for the programmer to consider: processes are taken as fundamental or primitive language concepts and building blocks. Hoare, in fact, saw the process as an object, expressed only in terms of its behaviour as an 'executing device'. He explored how processes could communicate and synchronise with each other.

If several processes can run at the same time, a mechanism must be proposed to do this. Hoare described a 'primitive' for parallel execution of processes (based on Dijkstra's *par-begin*), the semantics of which imply that the relative execution speeds of any component processes are irrelevant and further that this parallel command is not finished until all its component processes are finished, that is, the parallel command must wait for the last process to finish.

With parallel execution, the issue of inter-process communication is raised. Processes have to communicate for purposes of synchronisation or for information exchange (the exchange of data via the sending and receiving of messages between processes). Hoare explored how the input and output between processes are addressed.

Message Passing

In order for running processes to exchange information between themselves, Hoare proposed 'message passing'. We contrast this with how variables in sequential programs are traditionally updated—by some sort of assignment statement. He compared this mechanism with the needs of concurrently running processes and argued that, between processes, input and output is sufficiently different to warrant a conceptually different mechanism. He proposed two primitives: send and receive.

Definition

The syntax for receive is:
<fromProcess>?msg
and for send:
<toProcess>!msg.

Syntax

These commands are composed of three parts. First, a receiver names its sending process and a sender names the receiving process. The symbol "?" for the action "receive" and "!" for "send" is placed next, and finally, the actual message exchanged is placed last. The message can be a single variable or a structured variable as long as these types match between the sender and receiver.

Semantics

Hoare discussed using channels (port names) instead of processes as the first component in a send or receive command. He rejected this idea because he felt using processes was more readable. As well, the possibility of connecting more than 2 processes via a single channel added complexities that took away from the issues he wanted to deal with. PROMELA does use channel or port names and more than two PROMELA processes can use a single channel. In fact, the language allows any process to access *both* ends of the *same* channel.

Hoare defines his message exchanges in terms of synchronous communication. This

means that corresponding send and receive commands in different processes must be executed simultaneously. If one process arrives at its message exchange statement before the other, it must wait.

Asynchronous (buffered) communication is not a primitive notion for Hoare. To model asynchronous communication, as many processes as the buffering capability of the asynchronous channel to be simulated are used. Each buffer entry process alternates between reception from an upstream neighbour and sending to a downstream neighbour, so that the total number of component buffer entry processes defines the buffer size [6].

For Hoare, the alternative construct does not require at least one guard to be true: if all guards are false the alternative command simply fails.

Issues for Communicating Protocols

Hoare allows reception statements to be used as guards. This has dangerous consequences but largely represents exactly the types of systems we want to study when building communications protocol models. For example, if no sender completes an exchange where a reception is used as a guard, the possibility of 'deadlock' exists. This is an important issue to explore in any given model. This can be seen in the following scenario.

Using a repetition construct, with all guards being receptions, the only way for this loop to continue is to either wait for the first corresponding send to complete the exchange or wait for all source processes to terminate thereby terminating the loop. If neither of the two above events occur, the whole process stops, frozen in deadlock.

The issue of 'fairness' was also brought up: a choice can not be indefinitely not chosen and implementations rather than the programming language should guarantee fairness. Also, the programmer must show that the program terminates correctly. After considering parallelism and message exchange, Hoare concludes that input/output and concurrency should be primitives of any programming language.

3.3.5 Finite State Machines (FSMs)

The FSM is a fundamental model of computer science.

[6] This synchronous buffer concept is also a fundemental premise in th LOTOS language [BB87]

"The FSM model provides the foundation for consistency and completeness properties linking inputs, outputs, state, and processing, and provides the link into program proof techniques". [Lam85]

Formally a FSM can be viewed as follows:

- set of inputs, X
- set of outputs, Y
- set of states, S
- an initial state, S_0
- 2 functions that specify the result of inputs for all states at each state:
  - a transformation function that specifies the resulting output
  - a state transition function that specifies the resulting next state.

Its main limitations are that:

1. it serialises away all of the underlying concurrency ("clash states" are neglected)
2. the model is "flat": there is no inherent way of handling complexity

Figure 11 is an example of a FSM. These limitations are overcome by introducing "extensions"– essentially allowing variables and modules within the context of a programming language. FSMs that are used to describe communications systems are often extended by adding modules, variables, and send and receive primitives. Once a FSM system is designed, a trace is often used to describe the behaviour of these models.

3.3.6 Reactive Systems

Protocol models are part of a larger class of models called reactive systems.

Reactive systems are distinguished by the fact that they are always accepting input. Unlike sequential programs, they are highly interactive with their environment [MP92].

"The fact that the state of the system is changing while a process is viewing it is the fundamental problem in synchronising any kind of concurrent systems" [BBBC93].

As previously mentioned, sequential systems can be thought of as a function, where an input occurs, followed by its processing and finally its return. The fundamental concern of sequential programs is the input/output behaviour of the machine (the relation between the starting and terminating states).

Concurrent machines, whether shared memory multiprocessors (tightly coupled) or distributed systems (loosely coupled) are more complicated. With concurrent programs, two concurrent subprograms can interfere with one another. So one must look at the complete behaviour of the model: what the program does throughout its execution lifetime.

Formally, we can look at sequential programs as a special case of concurrent programs. First let us attempt to illustrate a representation of the execution of a sequential program:

$$\Sigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} s_3 \cdots \xrightarrow{\alpha_n} s_n$$

where,

- s_i are states
- α_i are atomic actions
- Σ is the sequence which is finite iff the sequence terminates.

This provides us with a simple trace that describes the behaviour of the system. Each state contains all the necessary information to determine its future behaviour. The sequence is termed "deterministic" if each state contains precise information about what its next move should be. The behaviour of the entire sequence is described as deterministic. Non-deterministic behaviour describes a situation where a given state has more than one choice.

Often, we wish to look at the set of all possible behaviours of a system due to its component actions. Actions are deemed atomic if an action, e.g. $x = x + 1$ proceeds uninterrupted from a current state where $x = x$ to a next state where $x = x + 1$.

How the program is divided into atomic actions is irrelevant for sequential programs but is important for concurrent ones.

3.3.7 Concurrency

Different formalisms are used to describe this behaviour trace. We can describe this trace as a sequence of actions (space-time view) or a sequence of states (the interleaving view).

The space-time view describes its execution model in terms of actions. Although actions within a process are totally ordered, actions across processes can not be viewed as sequentially ordered. The space-time view assumes a partially ordered set of events determined by events per process totally ordered in time and cause must preceed effect for inter-process actions. "This set of temporal orderings must be acyclic, so its transitive closure forms an irreflexive partial ordering" [Lam85].

The interleaving view reasons in terms of states. One constructs a global state-transition graph as per sequential models, in the sense that, if two actions occur at the same time, and they do not influence each other, we may pretend that they occur in either sequence. If we find that these actions do influence each other then we may say that the actions are NOT atomic so we may now increase the level of atomicity in the model. This, of course, requires that we lose precedence relations on a global level.

3.4 Validation and Verification

It is not enough to specify a system (although this can be a major task by itself). In order to be confident that the model reflects reality it must also be validated and verified.

Validation refers to checking general properties of the model in question. What we mean by the general properties are the characteristics that must be true regardless of the specific characteristics of any one protocol. Thus validation consists of checking for no deadlock, no unspecified receptions, no livelock, etc. [Lin85] defines validation as the properties that are apropos to all protocols and suggests that the following should be checked for:

- absence of deadlock
- completeness: ability to handle any condition that may occur
- absence of livelock: continual cycling of code that does nothing useful
- freedom from overflow

- stability
- proper termination.

Verification refers to checking the characteristics or properties that are unique to a protocol. [Lin85] defines verification as "checking the logical correctness of the protocol". For example, if any specific claims are made about the protocol, the proof that these claims are true is termed verification. In the language that we will use, specific PROMELA language primitives are available in order to do this.

3.4.1 Reachability Analysis

Reachability analysis is used to validate protocol models. The idea is conceptually simple and has been used for some time. Each single component module's behaviour is described via a FSM. In order to describe the behaviour of the entire system, each module and the interactions between them (as represented by the messages in transition for each channel) are used to yield a 'global' description. Reachability analysis attempts to build this complete system—essentially delineating all possible traces, which taken cumulatively, describes the global system behaviour.

The behaviour of the global machine can be represented by a finite graph, where every node represents a reachable system state and every transition represents the execution of a single statement in a single process [HGP92]. Once this graph is constructed a reachability analysis algorithm is used to traverse the graph in order to explore all the global system states.

The problem is that this methodology is often not practical because of the combinatorial explosion of the global state space, known as the 'state explosion problem'. As the complexity of the protocol increases, the number of states that have to be checked grows exponentially until it is too time or resource consuming to check the entire protocol model.

Graph Traversal Techniques

Different reachability algorithms have been designed through the years in order to make the size of the problem space that can be covered by reachability analysis larger.

For example, one technique, the 'random walk validation' only analyses a small fraction of the reachable state space but it can detect a large fraction of the errors in a complex protocol, provided the errors themselves have limited complexity. The idea is based on the fact that in global FSMs there is much state duplication and any random walk down the many available traces will allow one to 'hit' most of the errors, if any. This method is described in [Wes89].

Different approaches to this problem are described by many other researchers. Holzmann [Hol85] and [Hol87] provide a good overview of these approaches. Holzmann [Hol90a] and [Hol90b] also provide many examples of algorithms for protocol verification. Holzmann [Hol88] introduces the bitstate approach, which is implemented in the SPIN tool. Holzmann et al. [HP89] discusses many of these issues for SDL. The authors also describe a way to pose correctness assertions directly in temporal logic, which curiously, has yet to show up in the SPIN tool.

[HGP92] describe three new reduction strategies for conventional reachability analysis. They describe implementations of partial order semantics rules that attempt to minimise the number of execution sequences that need to be explored for a full state space exploration. This has recently been implemented in the latest version of the SPIN tool [Hol95b]. They then go on to describe a method to minimise the amount of memory that is used to build a state space. They follow up on this last idea in [GHP92] where they maximise the number of global states that are cached in memory. They describe a method to avoid exponential increase of the run-time overhead caused by interleavings of a same partial ordering of statement executions leading to the same state.

As can be seen, the state explosion problem can not be avoided. Instead optimisations based on certain assumptions about the model are used to help design algorithms that make the analysis practical, yet still cover all the relevant states.

3.4.2 Temporal Logic

Logic is most often used to talk "about" a model in the protocols community. "The goal of reasoning about a program is to show that $\models A$ is true for a desired property A."[Lam85]

One uses linear time temporal logic [Pnu77, Hai82, MP92, Lam85], where actions are

ignored and only states used. Temporal logic uses several notions from classical propositional logic. Claims are expressed as assertions which are well formed formulas (wff) of classical logic and all new formulas are derived from temporal logic. [Got92, HG] provides a good introduction to temporal logic.

3.4.3 Classical Logic

Classical logic is defined as follows:

- P - atomic predicates
- logical operators:
  - or $\vee$,
  - and $\wedge$,
  - negation $\neg$,
  - $\supset$ or $\rightarrow$ for material implication

Formally a state is defined as a truth-valued function on the set of atomic predicates, which is written as $s \models P$, in order to describe the value of state s on predicate P.

Taking a more abstract view, we look at each state as being composed of many variables (including a program counter) and each state is determined by the sum total of the value of each predicate that it contains. Predicates are defined as all the variables taken singly or any logical combination of them.

From this we can build a model M: (S, Σ) where S is a set of states and Σ is a complete set of sequences of states (trace) satisfying property E starting at any possible state.

Property E assumes that all future behaviour depends on the present state and not how the present state was reached.

We use the following notation for sequences: any element s of $\Sigma : s = s_0, s_1, s_2, \cdots$ represents a particular state. If the sequence is finite $s_0, \cdots s_n$, then $\exists n$ for some n.

We also find it convenient to define

- for a finite sequence: $s_m = s_n, \forall m > n$ where s_i represents the state of a program at time i.

- for a sequence length > 1 define, $s^+ = s_1, s_2, \cdots$ so that
  - $(s^+)i = s_{i+1}$
  - $s^{+n} = s_n, s_{n+1}, \cdots$

E can be restated if $s \in \Sigma$ then $s^+ \in \Sigma$ as the set of all possible executions starting from all conceivable states.

The following shows how a temporal logic assertion is to be interpreted as a statement about a model; for any model (S, Σ and any $s \in \Sigma$, define what it means for an assertion A to be true for the sequence s. $s \models A$ means A is true for s for all traces. To be true for S, we write: $\Sigma \models A \equiv \forall s \in \Sigma : s \in A$ or we can write $\in A$.

Specific temporal logic operators are added to the classical logic to make the logic more expressive. Specific operators are created in order to express the properties we wish to prove about a model. Specifically these express properties that hold or will hold over time as expressed over the states of the model we build.

Although different systems have been built up using different bases thay can all be expressed in terms of the other.

Unary Temporal Operators

The duality of operators show that sometimes one can be expressed in terms of the other, i.e. they complement each other, for example, *henceforth* $\Box$ is the dual of *eventually*: $\Diamond \equiv \neg\Box\neg A$ and $\Box \equiv \neg\Diamond\neg A$ for any assertion A.

When talking about models, we express the model as a series of states, such that, we use the above operators as follows: $s \models \Box A \equiv \forall n \geq 0 : s^{+n} \models A)$

and for all states in the sequence we can say:

$s \models \Box A \equiv \forall n \geq 0 : s_n \models A)$

Eventually A or $\Diamond A$ can be defined as follows: $s \models \Diamond A \equiv \exists n \geq 0 : s^{+n} \models A$.

$A \Rightarrow B$ is a useful derivation which is defined as: $\Box(A \rightarrow \Diamond B$ which should not be confused with material implication from classical logic [7] which means if A ever becomes

[7] This is easy to do because different authors use different symbols for the same concept, or, worse, the same symbols for different concepts, and often define different useful derivations, often ignoring or dismissing results and working out entire systems of proofs without ever referring to alternative systems. Not too mention, the confusion that results from the many systems in use.

true then B must now or eventually also become true.

Binary Temporal Operators

Many binary operators are derived from the unary operators to simplify the expression of certain properties. All the binary operators are expressible in the terms of the other and for that matter using either $\Box$ or $\Diamond$, any one binary operator can be expressed in terms as the other. As defined earlier they $\Box$ and $\Diamond$ are dual to each other.

The binary operator that expresses the proposition that B remains true until A becomes false $A \trianglelefteq B$ asserts that B holds as long as A holds but if A becomes false at any time t then $A \trianglelefteq B$ can say nothing about the truth of B, in other words, from time t on (After A becoming false), B can be either true or false, we have no way of knowing, or we have no desire to impose this requirement on a model.

$A \trianglelefteq B$ is defined as follows:

$$s \models A \trianglelefteq B \equiv \forall n \geq o : \forall m \leq n : s^{+m} \models A) \rightarrow s^{+n} \models B$$

$\Box A$ can be expressed (and therefore be seen as a special case of this relation) as: $\Box A \equiv true \trianglelefteq A$. O More commonly used is the $\cup$ operator: $A \cup B$ or A until B is defined as $\neg B \trianglelefteq A$

A is true until B is true. (B supercedes A) after B is true we can nothing about A. This gives a natural way to show order of cause and effect.

$A \triangleleft B$ is similar to but allows if A becomes false, B will remain true at least one step longer than A.

As can be seen, temporal logic is a very expressive mechanism. It has a rich set of primitive and derived operators and rules for applying those operators to make assertions. Although we use it to help us verify systems, it has also been used as a specification language [MW84, MP92, Lam82].

3.5 Conclusion

In this chapter a survey has been made of some of the existing techniques used to describe and to test the correctness of protocols.

There are two styles of description: internal which describe the behaviour of the component modules and external which describe the observational actions of the system as a whole.

Informal methods were used to describe protocols before the advent of formal methodologies which relied heavily on natural language descriptions and their accompanying graphics. Formal methodologies allow one not only to describe but also to test for correctness of the description.

Amongst other techniques, we have examined the role of FSMs to specify protocols and the role of reachability analysis and temporal logic to test for correctness of the model. These are not all the techniques available but we feel that they are most appropriate for performing a logical analysis of a protocol specification.

There is a need for protocol development techniques in order to tame the complexity of designing and describing protocols. Such an approach is outlined in Appendix A.

All the models and approaches that have been discussed so far have been incorporated into PROMELA. Thus we find the language and its corresponding approach to be a very powerful tool to build and analyze protocol systems models.

Chapter 4

PROMELA, a PROtocol MEta-LAnguage

4.1 Why PROMELA?

In order to apply formal specification and validation techniques such as process algebra, Petri Nets [Pet81, Rei85, Rei92], automata, temporal logic, and reachability analysis to protocol development (not all were mentioned in the last chapter), languages have been created such as Estelle [BD87, CA90a, DAC+89, ISOb], SDL [ST87, BH89, FA94], and LOTOS [BB87, DB92, ISOc]. The above languages are established as international standards. The following references contrast them [MGK91, Tur93, Tar91, ISOa]. Among the plethora of languages available, PROMELA is the one that we feel best incorporates the most relevent techniques and which we, correspondingly discussed in the previous chapter.

PROMELA is a language that is used to aid in the design and validation of computer communication protocols. It is a hybrid language using C's syntax with the addition of Hoare's CSP (Communicating Sequential Processes)-like constructs, and some isolated constructs for use by validation tools (based on reachability analysis and temporal logic), with the added paradigm of the extended finite state machine model. PROMELA is presented in Holzmann[Hol91]. Its name is an acronym of PROtocol MEta-LAnguage[Hol93].

The language has two major characteristics: it can be used for specification and for validation. The language also has extensions to support program verification.

4.2 The Language

In PROMELA, extensions to the syntax are minimised in favour of using familiar constructs from its parent language C, with slight modifications. The number of operators and keywords is quite small. Many operators, keywords, and forms are inherited directly from C. Tables 1 and 2 list the language's keywords and operators.

Table 1 is a listing of PROMELA keywords sorted by keyword (the first column). with a short explanation provided in the third column. If the keyword is a direct import from C it is marked with a 'C' in the 'Lineage' column. The 'version' column signifies in which version the keyword was first introduced.

Table 2 follows a similar structure, except that the explanation is continued into the 'Type' field. This field is used to give an indication of how the operator might be used. Several operators are 'overloaded', in the sense that the same operator has a different meaning depending on the context in which it is used. Essentially all the operators are borrowed from C except for the 'message passing' types. Any other language components are either comments, (goto) labels, or user defined variables. Table 3 gives a brief description of the syntax for the remaining language components (comments, variables (identifiers) and the labels used in goto transitions) using a BNF format.

4.2.1 C Extensions

PROMELA also has access to such C-isms as:

- structures, to build records
- auto-increment (`var++`) and auto-decrement (`var--`) variables
- conditional expressions in the form: `(expr -> expr1 : expr2)`
- C pre-processor constructs such as #define and #include.

A concise look at the language and its role in protocol design and validation can be found in [Hol93] . The paper also contains a BNF-like syntax description for the entire language.

PROMELA Keywords			
Keyword	**Version**	**Explanation**	**Lineage**
active	2	initialisation	
assert	1	verification	
atomic	1	specification	
bit	1	type declaratrion	C
bool	1	type declaratrion	C
break	1	specification	C
byte	1	type declaratrion	C
chan	1	specification	
do	1	specification	
d_step	2	specification	
else	2	specification	C
empty	2	queue test	
enabled	2	verification	
fi	1	specification	
full	2	queue test	
goto	1	type	C
hidden	2	specification	
if	1	specification	
init	1	initialisation	
int	1	type declaratrion	C
len	1	queue test	
mtype	1	data type	
nempty	2	queue test	
never	1	verification	
nfull	2	queue test	
od	1	specification	
of	1	specification	
pc_value	2	verification	
printf	1	output	C
proctype	1	specification	
run	1	specification	
short	1	type declaratrion	C
skip	1	specification	
timeout	1	specification	
typedef	2	type declaratrion	C
unless	2	specification	
xr	2	queue test	
xs	2	queue test	
_pid	2	verification	
_last	2	verification	

Table 1: PROMELA Keywords

PROMELA Operators			
Operator	**Version**	**Explanation**	**Type**
':'	2	separator	conditional expression
';'	1	semicolon	command separator
'-'	1	negation	arithmetic
'-' followed by '>'	1	semicolon	command separator
'-' followed by '>'	1	separator	conditional expression
'-' followed by '-'	2	auto-decrement	arithmetic
'+'	1	addition	arithmetic
'+' followed by '+'	2	auto-increment	arithmetic
'<'	1	less than	relational
'<' followed by '='	1	less than or equal	relational
'<' followed by '<'	1	left shift	bit
'>'	1	greater than	relational
'>' followed by '='	1	greater than or equal	relational
'>' followed by '>'	1	right shift	bit
'='	1	assignment	assignment
'=' followed by '='	1	equals	relational
'~'	1	not	bit
'!'	1	not	logical
'!'	1	send	message passing
'!' followed by '!'	2	sorted send	message passing
'!' followed by '='	1	not equals	relational
'?'	1	receive	message passing
'?' followed by '?'	2	random receive	message passing
'&'	2	and	bit
'&' followed by '&'	1	and	boolean
'|'	2	or	bit
'|' followed by '|'	1	or	boolean
'*'	1	multipication	arithmetic
'/'	1	division	arithmetic
"('	1	start group	grouping
')'	1	end group	grouping
'['	1	start index	index
']'	1	end index	index
'['	1	start test channel	message passing
']'	1	end test channel	message passing
'{'	1	start process	process
'}'	1	end process	process

Table 2: PROMELA Operators

Miscellaneous PROMELA Syntax	
Regular Expression	**Type**
`/ * <comment> */`	comments
`(letter||underscore)(letter||digit||underscore)*`	identifier
`identifier :`	label

Table 3: Miscellaneous PROMELA Syntax

4.2.2 Model Specific Constructs

In PROMELA, the major influence seems to have been Hoare's Language of Communicating Sequential processes, which in turn owed a great deal to Dijkstra. The most interesting features of PROMELA: guarded commands, non-determinism, and multiple processes, are features directly imported from CSP, with minute but important semantic differences. The changes allow processes to be much less stringently synchronised (by buffering channels), and provide simpler means of re-synchronisation by blocking on guarded commands (rather than failing on error if no guards are runnable).

PROMELA eliminates facets of standard programming languages that are not appropriate in protocol specifications. For instance, pointers are not appropriate to (loosely coupled) communicating systems that do not share memory, so they are simply not present. In addition, since side-effects pose problems for verification systems, "temporal claims", which are used in verification proofs, are restricted to being demonstrably pure. Functions are not implemented in PROMELA, as everything can be done by a proctype module.

Communication ports, concurrency, and finite state machines are concepts which are invaluable and integral to any protocol specification language, but are not inherent to the C language. Communication ports and concurrent systems are necessary to protocol models because protocols generally deal with communicating concurrent systems. Non-determinism is a useful concept for modelling parallelism and concurrency control.

4.2.3 Guarded Commands

Guarded commands are PROMELA statements whose ability to complete successfully guards (prevents the execution of) the succeeding statements. In the following fragment, the printf

statement is guarded by the channel input request. The buffer must be filled before its contents can be printed.

```
Input?buffer ->  printf( "%d\\n", buffer );
```

When a guard fails, a process blocks unless there is some other action it can perform while waiting for the guard to succeed. This introduces the selection statement, and with it, non-determinism.

The *if* and *do* clauses imported from C consist of a number of guarded commands followed by their guarded code. If, during execution, one of the guards can be executed when the *if* statement is encountered, then that series of commands is executed. On the other hand, if multiple guards will complete successfully if executed, then any one of the executable streams may be taken. This is how non-determinism is supported by PROMELA.

The concept of the guarded command is not found in sequential programming because flow of control and executability are not at issue, in the sense that the program counter is always at the "next" statement and the condition of executability of the next statement is not dependant on anything other than this. With concurrent programming this is not true, as several guards may be true at the same time or a guard may be forced to *wait* for an event to become "true" in order to continue if no guards are presently "true". Compare this to sequential programming: there is no wait—a condition is either true or false. This same situation is seen with Hoare's semantics, which insist that there must always be at least one true guard. However, with concurrent systems—because there is communication—it is possible to change a value from "false" to "true" at a later date. The obvious question is how long do you wait, because the event may come true or it may not.

4.2.4 Timeout

"How long to wait" is answered by imposing a set period of time to wait and after that time has expired, 'timeout'. However, PROMELA does not simulate time, so one must use "timeout" as an abstract concept, in keeping with PROMELA's paradigm of performing a

logical analysis. In keeping with this idea, the *timeout* keyword is introduced to allow an escape from a "deadlock". If no other transition can fire and a *timeout* is specified in a module, it will fire. The *timeout* condition only becomes true when the entire system locks up.

We note that a timer can be simulated in ones model by declaring a boolean variable 'timeout'. A timeout period for this timer is irrelevent in a logical analysis , thus it randomly fires—to simulate a logical timeout (without the use of the keyword 'timeout'). To show that a timeout occurs, one chooses non-deterministically between 'the timeout has occurred' or 'it has not'. Instead of trying to optimise and prevent a timeout from occurring, PROMELA allows timeout conditions to occur so that they may be dealt with. We want to explore *all* conditions and recover from them so that the timeout conditions are not optimised away. PROMELA performs a logical analysis, not an optimisation as in a pure simulation language like SMURPH [GR92b, GR92a].

4.2.5 Finite State Machines

FSMs in PROMELA At first glance, finite state machines seem to be somewhat of an afterthought in the design of PROMELA. For example, states and state transitions are not primitive constructs in the language: state transitions are implemented via changing the flow of control in a module via the *goto* statement jumping to a targetted line label (which is directly inherited from C without change). So, although FSMs may be the basis of the language, this primitive construct is not as apparent to the user as in other FDTs, such as Estelle.

This is somewhat disconcerting (but in keeping with its minimalist design) since finite state machines are used so heavily in the design techniques. For example, FSMs are used as a design and verification aid for individual processes, and to build complete models of the state of a given simulation of a protocol. Such models are used to perform reachability analyses on protocols, to help in verifying correctness and in judging completeness criteria.

Despite PROMELA's minimalist support for FSMs, FSMs are used and are an important part of designing models in this language.

Non-Determinism

Non-determinism is a useful technique for creating protocol models with finite state automata. For example, in a traditional finite state machine, when a given input fires, a single transition takes place which leads to another state. Sometimes the destination state can not be uniquely determined. When the choice among the states is not specified within the model but determined at run time, this is known as non-determinism. Non-determinism can be used internal to a module but is most often used when modelling the interaction amongst different modules in a model. As we have seen, however, this often leads to an explosion of possible states to explore.

Extending Finite State Machines

EFSMs One of the reasons that traditional deterministic finite state machines are not very expressive is that they have no concept of memory (as implemented by variables in programming languages), and it is tedious to create D-FSM's for useful problems (as seen in Valira [CA90b]). FSMs extended with variables are an essential ingrediant of PROMELA's marriage of FSMs to the C programming langauge. If one compares this to the traditional finite state machine approach, values of variables can now be used to represent entire groups of states, reducing the total number of states (and therefore the complexity) of a design.

CFSMs In order to model protocol systems with FSMs, EFSMs are further extended by adding channels as primitive language constructs. Concurrency is thus achieved by using multiple extended finite state machines, which communicate with each other via channels. Each CFSM is enclosed as a module unto itself, and has its own execution thread. Thus the overall structure of this system model is one of modules containing extended state machines that communicate with each other via channels. So the fundamental model of specifications in PROMELA is of groups of non-deterministic, sequential processes that communicate with each other via channels.

4.2.6 Execution Scheduling, Channels and other Aspects of PROMELA

PROMELA has no hierarchy of execution times, nor is there any subdivision for execution scheduling size. All synchronisation is done explicitly using channels and guards. The

PROMELA *proctype* processes are non-deterministic and sequential, and all compete for execution on an equal footing.

PROMELA processes execute in parallel, thus there is a need to delimit critical sections, when groups of actions that must be performed as a unit. The *atomic* construct is provided for that purpose.

PROMELA channels are of finite length (as compared to Estelle's 'infinite' length queues). Queue lengths can be used as a means of synchronisation (for example, forcing rendez-vous style with a queue length of 0). Channel I/O commands block on empty or full channel reads and writes respectively.

PROMELA has no concept of shared queues. However, any process which knows the name of a channel can access it, and will race (in the absence of other synchronisation means) for the channel data. For example, a process which sends a message along a channel and then immediately listens for a reply could easily accept its own output as input!

4.2.7 Validation and Verification

PROMELA is more than just a specification language. The language provides many constructs for validating and verifying PROMELA specifications. In fact, Holzmann calls the systems that can be created in PROMELA "validation models" as if to stress the property proving aspects of the language. Conveniently, much of this checking is done automatically by the SPIN tool.

Validation Keywords

The keyword 'assert' is used to make assertions about specific charactoristics in the model. For example, to assert that the number of received messages is always less than or equal to the number of packets sent, one could enter the line " `assert(no_rec <= no_sent);` " somewhere in the model. This might be done in a separate monitor process which would run concurrently with the model or placed in strategic places within the model itself.

Other checks can be performed by using special labels in the code. These labels start with a certain keyword, are followed by the characters that are legal for identifiers and end in a colon.

An "end" label says that it is alright to end at the current instruction (instead of at the last

statement in the module). This would be used, for example, to define a process that never ends—or if it did, this would be an error. In this way an operating system or some sort of system daemon could be modelled and then verified.

A "progress" label is used to signify a statement that must be passed in order for the protocol to advance. This would be used to check that a system is not trivially correct, in the sense that if a module never executes it certainly can not produce an error but a module that "does nothing" is usually not what the designer had in mind.

The "accept" label may be put in front of a sequence of commands that can not be repeated "infinitely often". This could be used to allow a sequence to execute not more than once. However, this label is most often used in conjunction with the "never" keyword in order to express temporal logic claims using Büchi automata, or as Holzmann calls them, temporal claims.

Verification Keywords

A Büchi automaton is an FSM that "accepts" only cyclic executions that contain at least one accepting state within the cyclic part (as apposed to a normal FSM that "accepts" only non-cyclic executions that lead it into an accepting state).

To express correctness properties for liveness in SPIN, the procedure is as follows [Hol95a]:

- express the negation of the property in next-time-free linear time temporal logic (LTL). This property formalises all counter-examples to the original correctness claim.
- build the Büchi automaton that formally corresponds to the LTL formula just constructed. The Büchi automaton is constructed such that it will accept precisely the counter-examples formalised in the first step.
- describe the automaton in the syntax of a Promela never claim, and add it to the specification.

It is not a trivial matter to express correctness claims in terms of temporal claims, so this is used 'rarely'. The problem is that, in the user community, most correctness claims are expressed in temporal logic, not in temporal claims, and as such it is not as easy to convert this type of assertion.

Converting Temporal Claims to Temporal Logic

An attempt was made to write a translator to convert from temporal logic to temporal claims but its output was deemed "buggy" [Gla94]. It is not clear what buggy means, is it not consistent in producing the correct answer or are there certain conditions that the code does not work?

The algorithm does not correctly translate a certain class of temporal formulae. These formulae are of the form [] <> p. We show the results of some of these translations on the next page. The tool is helpful but clearly some more work needs to be done here and this is on the agenda for improvements to the SPIN tool. We may see a more reliable translator or the ability to express correctness claims directly in "temporal logic".

Here are some example translations provided by the 'tl2nc' tool:

```
1> tl2nc
p
Old = p
New = p
never {
  state0:
            if
            :: p -> goto done
            fi;

  done:     T
}

7> tl2nc
[](p=><>q)
Old = [](p => <>q)
New = [](~p | <>q)
never {
  state0:
            if
```

```
            :: ~p -> goto state0
            :: q -> goto state0
            fi;

  done:     T
}

13> tl2nc
[](p=>(qUr))
Old = [](p => (q U r))
New = [](~p | (q U r))
never {
  state0:
            if
            :: ~p -> goto state0
            :: r -> goto state0
            :: q -> goto state1
            fi;
  state1:
            if
            :: (q & ~p) -> goto state1
            :: (r & ~p) -> goto state0
            :: r -> goto state0
            :: q -> goto state1
            :: (r & q) -> goto state1
            :: (q & r) -> goto state1
            fi;

  done:     T
}
```

Fortunately, Holzmann (quoting Manna and Pnueli) claims that only three temporal logic forms are useful in the context of protocol analysis. He gives the temporal claim translation

for all three.

The following is from [Hol93] and [MP90]. The explanations are from [MP90] and the corresponding translations into PROMELA code are from [Hol93].

Specifying Temporal Claims

> "There are three classes of properties we ...believe to cover the majority of properties one would ever wish to verify."

1. Invariance

> "An invariance property refers to an assertion p, and requires that p is an invariant over all the computations of a program P, i.e., all the states arising in a computation of P satisfy p. In temporal logic notation, such properties are expressed by $[]p$, for a state formula p."

Corresponding Temporal Claim in PROMELA:

```
never {
do
:: p
:: !p -> break
od
}
```

2. Response

> "A response property refers to two assertions p and q, and requires that every p-state (a state satisfying p) arising in a computation is eventually followed by a q-state. In temporal logic notation this is written as $p =><> q$."

Corresponding Temporal Claim in PROMELA:

```
never {
do
:: skip
```

```
:: p && !q -> break
od;
accept:
do
:: !q
od
}
```

Note that using $(!p||q)$ instead of 'skip' would check only the first occurrence of p becoming true while q is false. The above formalisation checks for all occurrences, also future ones. Strictly seen, therefore, the claim above uses a common interpretation of the formula, requiring it to hold always, or: $[]p => <> q$

3. Precedence

> "A simple precedence property refers to three assertions p, q, and r. It requires that any p-state initiates a q-interval (i.e. an interval all of whose states satisfy q) which, either runs to the end of the computation, or is terminated by an r-state. Such a property is useful to express the restriction that, following a certain condition, one future event will always be preceded by another future event. For example, it may express the property that, from the time a certain input has arrived, there will be an output before the next input. Note that this does not guarantee [require] that the output will actually be produced. It only guarantees [requires] that the next input (if any) will be preceded by an output. In temporal logic, this property is expressed by $p => qUr$, using the unless operator (weak until) U".

Corresponding Temporal Claim in PROMELA:

```
never {
do
:: skip /* to match any occurrence */
:: p &&  q && !r -> break
:: p && !q && !r -> goto error
od;
```

```
do
::  q && !r
:: !q && !r -> break
od;
error: skip
}
```

Strictly again, this encodes: $[]p => (qUr)$ To match just the first occurrence, replace skip with $(!p||r)$.

Thus if any properties of a model are to be verified this property can be expressed in one of the three ways just described (invariance, response, or precedencs).

Matching Temporal Claims

Once a property is expressed as a temporal claim, the FSM for the claim is run in parrallel with the FSM for the model. In order to prove the property true, we want to show that the "claim" does *not* become true. A never claim is "matched" as follows:

- if a condition is NOT explicitly stated as a GUARD you can back out of trying to match the temporal claim for that execution sequence (claim not violated and SO FAR correctness claim OK)
- if a match on a condition occurs, continue trying for another match UNLESS the end state is reached or you come to rest at an accept label
- if you can cycle back to it at least once, the temporal claim is matched, therefore a path exists that defies the correctness claim, therefore the correctness claim is FALSE

4.3 PROMELA Tools

Presently, there exists only one tool for PROMELA users, called SPIN. However, we have created two tools: one tool for creating time sequence diagrams and another for drawing finite state machines from PROMELA models. We will discuss SPIN and its X windows version XSPIN in this chapter. The tools for creating TSDs and drawing FSMs will be discussed in later chapters.

4.3.1 SPIN Usage

The main usage of SPIN is as a simulation package and as a validation package.

Overview of Methodology

The first step in the verification process is to write a PROMELA model. One usually then starts with a quick random simulation of the model in order to check for obvious correctness claims violations (deadlock, unspecified receptions, assertion violations).

If no errors are reported a fixed seed simulation with the send and receive (-s -r) options enabled to SPIN should be made. This provides for further information about the model. Standard Unix tool like "grep -v" can be used for filtering out needed information that this generates. Printf statements can be inserted into the code to be used for debugging or just to gather more information about the model.

If the results look reasonable one can then create and run the analyser for the validation of the model.

If the model passes the validation tests and, if verification of specific properties are required, a temporal claim that expresses the opposite of the property can be added to the protocol model (but the printf's should be removed from the code first).

To summarize, the steps to follow are:

1. build a protocol model
2. create a validation analyser: spin -a <PROMELA model file> see table 4 for the complete set of options
3. compile the produced C code (to produce a valiator) using the options listed in table 5

4. run the produced binary using options listed in table 6

5. look at the validator's (usually pan) output.

[Hola] is a user's manual for SPIN and [Hol95b] is an update addendum which discusses the new language and tool features that were introduced after January 1995. [Hol93] outlines how SPIN is usually used to validate models and [Holb] is a short paper that discusses in detail the steps required to follow when validating a PROMELA model in SPIN.

4.3.2 Simulation

In the concurrent model that we employ, it may be recalled that, at any given time, several processes may be eligible to run, and in a single process, several events may be eligible to fire. Which process and which event will be picked is not deterministic. SPIN has a simulator that acts as a process scheduler by continually picking one event from one process until either an error is detected or until each process runs to its final end state. For a single system, many choices are often available at any given time so that when a system model is simulated this often results in many possible sequences of events and state transitions. This "trace" is not always easily reproduced.

A simulator takes a concurrent model, consisting of an arbitrary number of executing processes and is used to produce an output trace. Each step in this trace represents the execution of a single process and its corresponding event that fired; this contributes to the evolving global model. As each step is taken, the simulator can be instructed to output useful information such as a process' variables or the occurrence of a send or receive event. See table 4 for a list of these options.

Seeded Simulation

The user can choose to produce either a random or a guided simulation. Random simulations can be "controlled" by using a 'seed' value. The seed insures that the same trace is followed each time the model is simulated. Thus the same 'random' simulation may be repeated if the same 'seed' is used in other simulation runs. This is useful when one wants to examine the same trace at a different level of detail and one wants to guarantee that exactly the same trace will be reproduced. SPIN is rerun with the same seed value but different options from table 4 are used.

Spin Command Line Options use: spin [-option] ... [-option] file	
Option	**Explanation**
-a	produce an analyzer
-d	produce symbol-table information
-D	write/write dataflow
-D -D	read/write dataflow
-g	print all global variables
-i	interactive (random simulation)
-l	print all local variables
-m	lose msgs sent to full queues
-nN	seed for random nr generator
-p	print all statements
-r	print receive events
-s	print send events
-t	follow a simulation trail
-v	verbose, more warnings
-V	print version number and exit
-X	internal, signals 'xspin' usage

Table 4: Spin Command Line Options

Random Simulation

The simulation can be different every time if no seed value is chosen. Without an explicit 'seed' the simulator constructs a random trace.

Guided Simulation

SPIN can be forced to follow a 'guided simulation'. A special 'trail' file is produced as the result of the validator finding a sequence of events in error. The user then uses this trail to debug the protocol design by following the trace trail and (hopefully) recognising what went wrong. The protocol can then be changed and re-examined with SPIN.

Interactive Simulation

Recently an interactive option was added to the simultor, allowing the user to control the scheduling of events and process that will fire. We have created a tool to record and play back these interactive traces, called 'exp.awk'. This tool gives us much more control and insight into the model when working with simulation traces.

4.3.3 Validation and Verification

The second function of SPIN is to act as a validation and verification tool. Using SPIN's analyser option, produces C code which can then be compiled and run. This C code is a custom-made validation tool, optimised and written specifically for the protocol model being examined.

Various customisation options are available when working with SPIN. When compiling the SPIN produced C files, several options to the C compiler can be invoked. (see table 5) The resulting binary file also has a standard set of flags available for use (see table 6).

The composition of all the running single processes forms the global state machine. A global state machine may be built first and then traversed or it may be built 'on-the-fly', which is what SPIN does.

Different validation strategies are supported and these can be controlled by using command line options. A partial serach using the bitstate algorithm or a full state search is supported. As well, a partial reduction algorithm can be used with either to produce better coverage. For a partial search, unreached states indicate coverage. For a full search, unreached

C Compiler Command Line Options note: use same C compiler eg cc or gcc use: CC [-Doption] ... [-Doption] FILENAME	
Option	**Explanation**
BITSTATE	use bitstate algorithm
CHECK	debugging
DEBUG	debugging
MEMCNT	explicit memory size request
NOBOUNDCHECK	no array bounds checking
PEG	complexity profile
REDUCE	use reduction algorithm
VERBOSE	debugging
VERI	debugging

Table 5: C Compiler Command Line Options (for compiling the analyzer)

states indicate dead code.

4.4 Conclusion

PROMELA allows the designer to write formal, unambiguous descriptions of protocol systems and then to validate those models in a simple and easy to use syntax. It is a sparse yet powerful language.

SPIN implements a compiler for the PROMELA language. XSPIN is its X window interface. Good as they are, they are somewhat wanting in their graphics capabilities. In the following chapters, we will elaborate on ways that this can be remedied, as we introduce two tools that we have developed.

Verifer Command Line Options note: assume verifier named "pan" use: pan [-option] ... [-option]	
Option	**Explanation**
-a	find acceptance cycles
-cN	stop at Nth error (default=1)
-d	print state tables and stop
-d -d	print un-optimized state tables
-f	enforce weak fairness (with -a or -l)
-i	find non-progress cycles
-mN	max depth N (default=10k)
-n	no listing of unreached states
-s	forward single-bit hashing (iso double-)
-z	backward single-bit hashing
-wN	hashtable of 2^N entries (default=22)
-RN	repeat Nx (1..32) with different hashfcts
-V	print SPIN version number and exit
-X	internal, signals 'xspin' usage

Table 6: Verifier Command Line Options

Chapter 5

Graphics support

5.1 FDTs and graphics

In the past, methods were used to describe protocols in the most intuitive way that humans know of, that is, by using graphics and text. Formal methods were introduced to make these descriptions more precise (less ambiguous). Unfortunately formal descriptions tend to be difficult to understand.

We explore the generation of graphics from formal models, thereby combining a very effective informal technique with the precision of formal methods. Using a formal model as a base, we not only have an unambiguous description of our model, but we can represent this information in a "user friendly" way. The graphics are completely consistent with the code because they are generated *directly* from it.

5.1.1 Characteristics of Graphics

Graphics are intuitive in that they quickly provide lots of information in a small space which we can easily assimilate. We humans are good at pattern recognition and can see relationships between objects just by looking at a picture. The evidence from evolutionary biology demonstrates this—and the fact that informal methods rely so heavily on graphics would tend to bear this out (see the quote in subsection 1.1.2 in the introduction).

New offers many arguments supporting the benefits of using graphics in protocol design (see [New91] for this justification and the many psychological references that he provides

to support this thesis). In one example he contrasts "spatial proximity" with "formal proximity" in order to show that a graphic can show overall model structure more easily than a text based structure. Simply put, one can see the graph connectivity (and thereby the overall architecture of the model) much more clearly when it is drawn out than when it is listed in a table or a program listing (see the chapter on FSM's (chapter 7) for some examples).

To emphasise this point, New provides and contrasts text and graphical based descriptions of a large protocol model. The text version is long and the information can only be sequentially unravelled as one reads through the listing. The overall structure is not easily apparent from this format. The graphical version, on the other hand, displays the overall architectural structure (what is connected to what) very clearly in the space of a single page.

Using graphics in protocol design builds on our intuitive, human evolutionary skills to recognize patterns, with the added benefit of producing a "formal" graphic being based on a sound, formal model. Graphics tools guarantee consistency between the model and graphics but should be easy to use. Using a graphical approach it is possible to focus on the specific aspects of the protocol we wish to study. If one does not have to concentrate on how to draw graphical linkages (but allow the user to intervene), the user can focus on the high level design and not be concerned with extraneous details. This allows the user to focus on the "important" things in the model.

5.1.2 Advantages of using graphics

Graphical representation of protocol models are very useful because they help the designer literally and figuratively 'see' what is going on. This is an informal mechanism; it adds nothing to the formal description; besides, we can not *prove* this claim for the usefulness of graphics. We could perform psychological tests or measure programmer productivity and maybe even "user satisfaction"— to a degree, but we can not prove this in a 'mathematical sense'. However, after having used these techniques we can make the following observations about them.

A graphic representation is very useful in the beginning stages of the design process, as the model is being decided upon or later as the code is being debugged. After the specification is completed the graphic can be used as a pedagogical tool for people unfamiliar with the protocol (users or implementors) who wish to study it. This last point is important as

this reduces the learning time of a protocol for a new user.

What graphics support does for formal descriptions is to add, quickly, correctly, and at a very little cost, a very useful aid for humans. If the graphics support can be used in conjunction with formal methods, the best of both worlds is realized. If the graphics support can be automated with the help of a computer, then we have a very powerful tool at our disposal. New calls this approach "protocol visualisation"—the combining of the strengths of the human, computer, and specification techniques in order to design protocols. Figure 2 in subsection 1.1.2 illustrates this three pronged approach to protocol design.

Our approach has been much influenced by New's work on his Estelle based "GROPE" tool. We have explored these ideas to see what advantages we could derive from representing certain aspects of the protocol model and design process graphically. Another goal is to develop algorithms or, better yet, tools that could then be used to automate a graphical representation of a protocol model. The following sections discusses some of our findings.

5.2 Graphics Support Tools

Tools are useful but they can not do everything. Graphics tools often require some level of user interaction, because, "the primary goal of a protocol visualisation tool is to use a computer to assist a user in understanding a formal protocol specification." [New91]. That is, the tool should not take over the entire design process: these tools can be used to do most of the tedious work, but the user must remain in control.

A graphics drawing tool will typically have a data extraction phase followed by a drawing phase. There is a lot of information that can be distilled from a formal model, so the first phase of a graphics tool must deal with extraction of the relevant structural data. The major task here is to produce an algorithm (or algorithms) that can use the information taken from the model and calculate a way to draw or represent this information graphically.

Once that is done, the second major phase, the actual drawing phase can start. The drawing tool then applies the data produced from "layout algorithm" and draws the graphic.

5.2.1 Dynamic and Static Graphic Information

Graphics information can be either dynamic or static; some graphic information can only be deduced from "running" the specification whereas other information (for example, architectural information) can be derived from a static model description. Dynamic information can be stored with the instantiation of an object in the model, whereas static information can be stored with the object's definition; thus, the question of storing the graphic information with either the instantiation of or the actual object is important one to answer.

5.2.2 Types of Graphic Representation

Another issue to be resolved is choosing what kind of graphics to support. Once chosen, the next question to answer is what is the best way to layout the objects and what sort of graphics to use for each object.

We can see three useful types of graphics for protocol models:

- the overall architecture of the model,
- the "external" view, based on time sequence diagrams and
- the "internal" view, based on finite state machines.

The overall architecture gives the user a general overview of the system. In our model of a system we use modules that represent certain functionalities which are then connected by channels carrying messages between the component modules. This provides us with a global view of the model; the modules at this point are "black boxes" but the overall connections are clear. Figure 14 is an example of the architecture of a model for the 'InRes' protocol. One can see that this small figure provides a lot of information as to how the overall system is "put together".

The other two views can be explained relative to this system architecture. In the internal view of the architecture, ("internal" means internal to a particular module) we can now start to define the "black boxes". A finite state machine is used to define a specific module's behaviour. In contrast, the"external" view provides us with a representation of the messages transmitted between the modules. We use time sequence diagrams to represent this view, so that the system is only defined in terms of its message transmissions and not in terms of the internal behaviour of its modules.

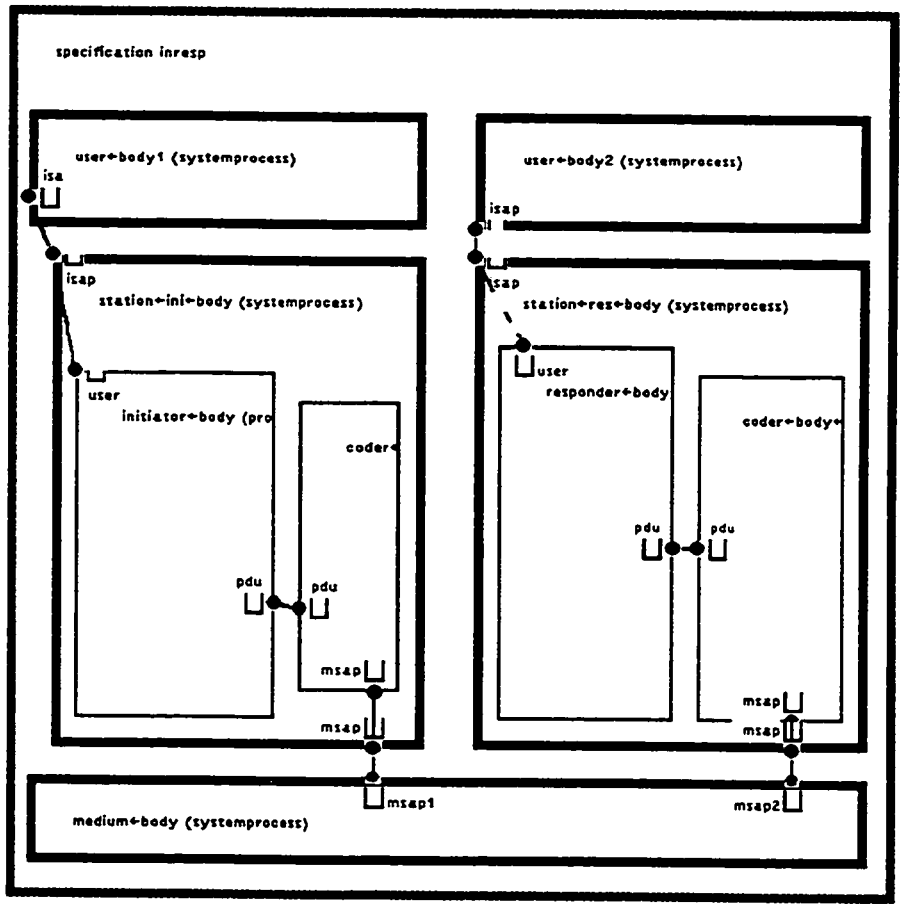


Figure 14: InRes Protocol Architecture

5.2.3 Languages and their Graphics Support

Previous to the start of our work, no graphic support existed for PROMELA users. Graphics support, however, exists for other formal specification languages. Some languages have a graphics version in addition to their "textual" version, whereas other languages may have a graphics component to one of its tools but not to the language.

For example, WISE and GROPE are tools for Estelle in the Smalltalk environment [New91], yet Estelle has no formally defined graphics representation. Graphics are deemed so important for other specification languages that there are graphic versions of them: G-LOTOS for LOTOS [BNT94] and SDL-GR for SDL [FA94].

Many languages that have a graphics component have to deal with how to represent the "extraneous" graphical information necessary for drawing the graphic but not usually part of "the language text". For example the LOTOS code has no provision for storing the graphical information that is seen in the G-LOTOS code. It is easy to translate G-LOTOS to LOTOS, but translating the LOTOS to the G-LOTOS representation is difficult because LOTOS has no provision in its text based description for "layout" information. This problem is also seen with SDL, which has a text version called SDL-PR and a graphics version called SDL-GR.

GROPE

GROPE [NA89b, NA89a, NA91, New91] relies on and enhances our natural cognitive and visual processing capabilities by offering a dynamic, graphical environment for protocol design for the Estelle language.

Background GROPE is built 'on top of' WISE which in itself is built 'on top of' SmallTalk. WISE, by itself, has no graphics and the WISE-GROPE system performs no validation. New does not experiment with any validation or verification tools, but adds graphics to an already existing simulation environment. The environment also includes a syntax directed editor for Estelle, called WIZARD.

New talks of the "advantages of resting user interface development on a firm formal base." and how the tool was written with the inter-operability with other Estelle tools in mind. Unfortunately, to date, no new tools have been produced. This may be because GROPE was written in an old version of SmallTalk and only runs on a Sun 3/50.

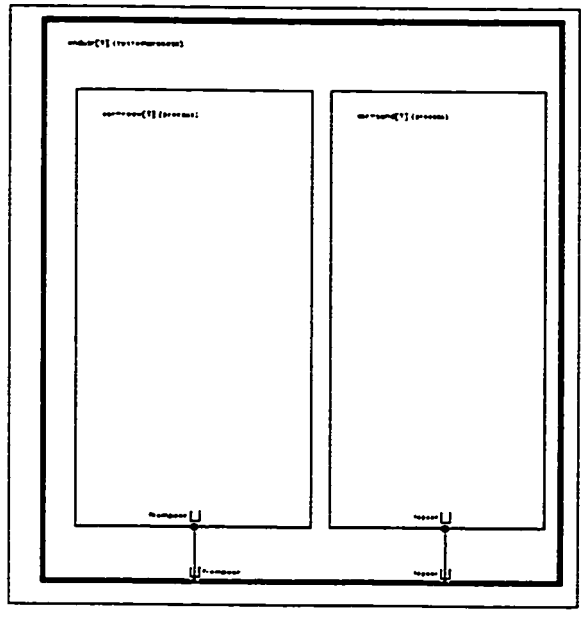


Figure 15: "End user"

Plans were made to port the package to the Next workstation and rewrite the package in Objective C. Unfortunately, we are not certain of the present status of the rewrite. But with Next no longer producing workstations and the low usage of Objective C (as compared to C++), if this port was completed, it would face as similar a 'dead end' as the Smalltalk version does now. This is unfortunate because the environment is an excellent one and the proposed system enhancements, such as being able to write the Estelle code visually would have been very useful.

Usage The graphical support that GROPE provides for the Estelle language comes in two types.

First, the overall architecture of the model showing the instantiated modules and their interconnecting channels can be easily drawn. (see figure 14 again, for an example of the architecture for the InRes protocol as specified in Estelle)[1]. Any part of the architecture may be 'zoomed onto' to show the internal architecture of a given module. This is seen in figure 15 for the architecture of the 'enduser' module of an early version of an XTP model.

[1]We used GROPE to find several errors in the "official" Estelle model of the InRes protocol, which we then sent back to the InRes designers for subsequent correction.

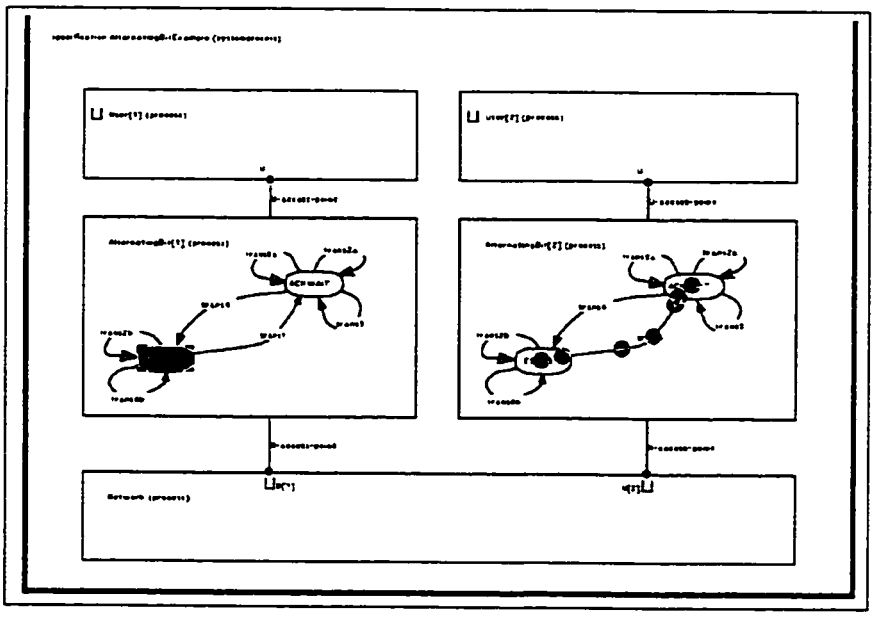


Figure 16: ABP: Multiple State Machines within the Model's Architecture

Second, the state machine of each module or process can be drawn at the click of a mouse button. (See figure 16 for an example of the state machines for the 'alternating bit' protocol as specified in Estelle). This state machine can fill the whole screen or can be drawn in the context of the model's architecture as shown in the last figure.

Protocols are dynamic systems but, as already pointed out, this temporal dimension can not be seen in a 'static' program listing. A major feature of this tool is that it animates a protocol model. GROPE represents message flow between the modules as an animation of 'black dots' that flow along their interconnecting channels. State transitions are similiarly depicted. By looking at the protocol 'in action', we can potentially see many design problems and eliminate them.

GROPE helps to bring out the 'cause and effect' that is seen when a protocol works, thereby allowing the protocol to 'come alive' and let the user experience this temporal dimension. New claims to have found bugs in long standing protocols and shows several examples that he found by simply animating the protocol models.

Simulating a Protocol Model The user can actually 'watch' as the protocol runs. A model is simulated and as the model runs, graphic windows illustrate what occurs. Among other

features, GROPE can highlight the current state per process and follow transitions between states and the sending of messages between modules. Allowing the user to watch the firing of a transition or the transfer of a message helps to give the user a very clear idea of how the protocol in question works.

Macro-states and Macro-transitions New investigates and supplies some visual simplifying algorithms. By grouping certain states together 'macro-states' can be created, as well as corresponding 'macro-transitions'. These simplifying capabilities reduce perceived complexities in the model and would benefit from further research.

Other Issues A particularly useful capability of the GROPE package allows the user to move the mouse to design the graphical layout which can then be saved to a file. Unfortunately the user must decide on the layout as GROPE offers no help in the calculation of a layout. The tool had some problems with printing its PostScript graphic files, but they were eventually solved by contacting the author of the tool.

Graphics Support for PROMELA

XSPIN is an X-based window interface to the SPIN program which provides all the functions of SPIN but through a more 'user-friendly' interface.

Version 1.6 of XSPIN was little more than a graphical 'front end' to the SPIN tool. It gives the user a convenient way to set the parameters normally set from the command line of SPIN or the analyser created from SPIN. In addition, a text-based linear list of all the states per FSM for each module in the model can be seen in a scrolling window.

In the 2.0 version of XSPIN many enhancements were added. For example, there are hyper-text links into the source code, TSDs are supported, and FSMs are drawn (but no resizing of drawing area (or 'canvas') is possible). The FSM graphic features display labels in the node and transitions; they switch from state number to line number when the mouse enters the node and PROMELA code is displayed when the mouse enters the midpoint of the transition arrow. The XSPIN package is much more useful as a design aid.

However, there are still some problems with the XSPIN package. For example the layout for the TDSs is not always correct and there is no way to change the layout of the proctypes or to intervene in the calculation of the layout. The user can change the layout using

the mouse after the graphic is drawn but can not save these changes to a file Further, if #includes are used in the source code, the proctypes defined in the included files can not be addressed correctly.

One nice feature is that message transmissions and proctype instantiations are dynamically created on the drawing canvas. Also there is a clear representation of when a proctype begins and ends.

5.2.4 Tcl/Tk

XSPIN was written using the Tcl/Tk package. Tcl is the language component and Tk is its X window graphics widget library.

The Tcl language has several attractive features. It is an interpreted Unix shell language: this means that programs written in it can be developed quickly, yet execution is very fast. Tcl is a very terse, yet powerful language ([Ous94, Wel95]). All its data types are essentially strings and associative arrays are supported in the language. All these features make it easy to write programs very quickly.

If we compare Tcl to Perl, its main competitor: Perl also has associative arrays but its syntax is very large. As well, Perl does not have graphics capability. So what made Tcl/Tk an attractive choice to use was its Tk component. The powerful X window programming library, with its full graphic support, is very easy to access using Tcl/Tk. (A Perl/Tk package called tkperl has at this time of writing just come out).

5.3 Our Tools

Our goal was to make our tools "filter-like" (in the Unix style of using component tools that do one thing in pipelines). As well our tools were designed to be as automatic as possible and yet allow the user to intervene and override them, if necessary. This would make them easy to use, even for people unfamiliar with them, as well as allowing us great flexibility when looking for what to change in order to ascertain what works the best. Another goal was to make "camera ready" figures that can be directly inserted into documents, at the same that we are using them "on line" when developing a model.

We decided to use Tcl/Tk because: [2]

[2]Our original plan was to use PostScript or the 'picture' environment of LaTeX to create our graphics. Initial

1. the XSPIN tool was written in this language

2. the graphics support is interactive and therefore dynamic information can potentially be displayed

3. the graphics support is at a very high level.

Our tools will be presented in the next two chapters. In designing our tools we wished to build on previous tools and also 'fix' problems we saw in others. A feature common to our tools is that if there exists a layout that the person likes, we do not recalculate a new one and we never over write over an old one.

experiments showed that LaTeX's picture format is too primitive, in that the basic objects are only lines, boxes and text. PostScript is a very good environment with a lot more functionality than LaTeX. Unfortunately, it too, is at too low a level for our purposes (one of which is to eventually include a dynamic component to the graphics), so we decided not to use it either.

Chapter 6

Drawing Time Sequence Diagrams

6.1 Introduction

We have created a tool we call "tsd" which produces data flow or time sequence diagrams (TSDs). TSDs document the external activities of a system model and by doing so, help the user develop a clear picture of the model under investigation. More specifically, TSDs show the sending and receiving of all messages transmitted between system components. (See the discussion under informal methods on page 19.)

We have prepared an ASCII version and an X11 windows version of the TSD generation program. (See Appendix B for the source code.) When referring to features common to both we shall refer to the tool as "**tsd**" or if this is not clear enough, "***tsd**". The X version is called "**xtsd**" and the ASCII version is called "**tsd**" [1].

Tcl/Tk was used to write XSPIN, and for that reason Tcl/Tk was also used to write ***tsd**. Our new functionalities could be integrated easily into the SPIN package or they could be used independently of XSPIN. For example, the **xtsd** functionality has been integrated into the latest release of XSPIN. This paper describes the 'stand alone' versions, which were designed to be run as 'filters'.

[1] I would have liked way to use one name, say "**tsd**" to invoke the program and let the system determine if its running over an X windows environment or not, but this was not possible to do within Tcl.

```
proc  5 (ReadingUserB)  line  54, Send LISTEN   -> queue 10 (outC)
proc  2 (context)       line  24, Recv LISTEN   <- queue 10 (fromR)
proc  2 (context)       line  31, Send Opencr   -> queue 9 (toR)
proc  5 (ReadingUserB)  line  57, Recv Opencr   <- queue 9 (in)
proc  5 (ReadingUserB)  line  60, Send WSync    -> queue 11 (SyncChan)
proc  3 (WritingUserA)  line  21, Recv WSync    <- queue 11 (SyncChan)
```

Figure 17: Sample Input (SPIN 1.6)

6.2 Background

We could see a use for TSD diagrams for modelling complex systems in PROMELA, long ago, but this functionality was not part of the original SPIN tool. However, we noticed that SPIN does output information that *could* be used to produce TSD diagrams by requesting that during a simulation run, SPIN produce information for each send and receive event that occurs in the system model.

SPIN simulations, when run with the **send** (-s) and **receive** (-r) options set, produce to standard output, information for every message sent and received. Figure 17 shows an excerpt from the output of a SPIN simulation run. Each line of output represents a single event of sending or receiving a message. By matching the receive for each corresponding send a line is drawn to represent the complete transmission of that message. It is also important to know if a sent message has not been received (message loss), although this is not shown in this figure.

Received messages can also be marked as "lost" when mimicking a channel that can distort messages. A message that was received in error is usually thrown away. As well, any messages that are received can also be randomly marked as "lost" as another way to mimic a channel that can lose messages. We'll see examples of this later.

TSD diagrams can be produced "by hand" from the relevant SPIN output lines. Arrows are used to represent the transmission of a message by connecting the appropriate sending and receiving modules together. This process can be extremely long and prone to error; as when, the user has to remember to match a send that occurred much earlier in the simulation; or when there are many modules, queues and messages involved in the model. This can

```
 7:    proc  5 (ReadingUserB) line  54 "./head.cm" Send LISTEN -> queue 10 (outC)
 8:    proc  2 (context) line  24 "./context.cm" Recv LISTEN   <- queue 10 (fromR)
11:    proc  2 (context) line  31 "./context.cm" Send Opencr   -> queue 9 (toR)
23:    proc  5 (ReadingUserB) line  57 "./head.cm" Recv Opencr <- queue 9 (in)
25:    proc  5 (ReadingUserB) line  60 "./head.cm" Send WSync  -> queue 11 (SyncChan)
26:    proc  3 (WritingUserA) line  21 "./head.cm" Recv WSync  <- queue 11 (SyncChan)
```

Figure 18: Sample Input (SPIN 2.3.3)

quickly become an overwhelming task.

6.3 Automating the Process

"***tsd**" was created in order to automate the production of this type of diagram. "***tsd**" *automatically* produces data flow or time sequence diagrams (TSDs) from SPIN simulation output and draws out the time sequence diagram accurately in a matter of seconds.

6.3.1 Input Formats

The ***tsd** tool can read three types of input. A command line option allows the user to choose the desired input format thus determining the specific input function used to read the input data.

The tool can accept input in these three formats:

- PROMELA pre-SPIN version 2.0
- PROMELA post-SPIN version 2.0.
- generic

A seperate parsing function is used to read each of the specific input formats just listed. All the rest of the code in the tool remains common. Therefore any new "language" can be easily accommodated by writing a parsing function for that language. As well, if the format for the output of a tool for a particular language is known it is also very easy to add a new input parsing function.

```
A AtoBq ! data
B AtoBq ? data
A AtoBq ! data
B AtoBq ? data
A AtoBq ! data||CNTL
B AtoBq ? data||CNTL
```

Figure 19: 'Generic' Input for **'tsd'**

```
A                B
A data           |
|                A data
B data           |
|                B data
C data||CNTL     |
|                C data||CNTL
```

Figure 20: Sample Output from **'tsd'**

We believe that ***tsd** versions for languages other than PROMELA, like Estelle, SDL, Valira and SMURPH would be extremely easy to write. For example, when version 2.0 of SPIN was released, we created a new function to read this new output format in a few hours. Figure 18 shows the corresponding format. A Valira extension would be especially useful as it relies solely on message passing.

As further proof as to the ease of extending this program to read other "languages" we created our own "language", a generic format. The generic format suggests the minimal information required to produce a TSD. As a side effect of creating the generic format we have found that we can use it to to draw time sequence diagrams that were not produced by SPIN. In this way we can illustrate potential message transmission scenarios or transmission sequences before we create an actual PROMELA model.

Figure 19 is an example of the generic input format. It depicts six events or three complete transmissions. This is the sending of two "data" packets from module 'A' to module 'B' and the sending of either a 'data' packet or a 'CNTL' packet (but not both) from 'A' to 'B'.

6.3.2 Output Formats

tsd and **xtsd** differ mainly in how they draw their message arrows. The way the arrows are drawn depends on the graphics capability of the output device (terminal or printer).

In the ASCII output of **tsd** the arrows have to be drawn in a 'virtual' way, a kind of "connect the dots". Instead of an arrow, a letter followed by the optional message contents

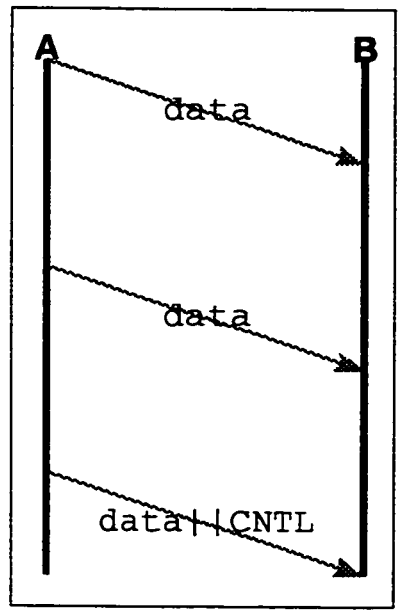


Figure 21: Sample Output from '**xtsd**'

is printed at the row and column that corresponds to the specific send or receive event.

tsd accomplishes this by entering these two events into a linear sequenced associative array, keyed by time, to be drawn out line by line after the whole model has been processed. Matching **sends** and **receives** are indicated by matching the upper case letters (see figure 20) because diagonal lines are impossible to draw on a "character" terminal.

xtsd simply draws a directed line or arrow between the two process from the 'time' or y-value of the sending event and the 'current' receive event (see figure 21).

6.3.3 Generic Input Format

The syntax for the generic input is most easily explained by re-examining figure 19 again. There are four columns: the module's name, the channel or queue used to transmit on, the type of transmission, and the message contents. The type of transmission may be either a send or a receive. The sending transmission is represented by the symbol "!" (or the word "Send"). The receiving transmission is represented by the symbol "?" (or the word "Recv").

The first column, the module's name, refers to the sending module if it is a send, or to the receiving module if it is a reception. There is no need for an "opposite" side in a transmission as the transmission events either put on or take off from the respective queue or channel

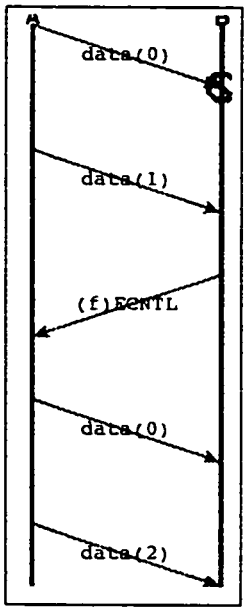


Figure 22: "normal" FASTNAK mode

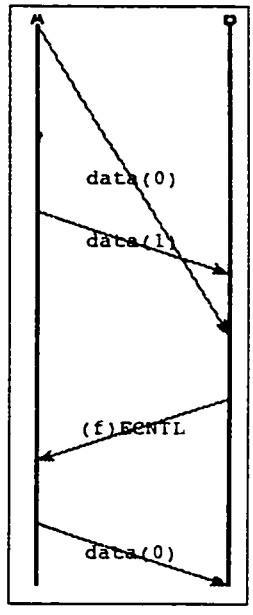


Figure 23: FASTNAK on a non-FIFO network

listed in the second column (this is also true for the PROMELA output).

Simulating Message Loss

There can be an additional column for reception events; in this case the user should set the LOSS_LINE variable in the tsd.h file in the current directory. If the value in this extra column matches the LOSS_LINE value then those receptions are considered to be lost message transmissions.

Message loss can also be more traditionally represented by an unmatched send, but this implies that there are to be more transmissions on this queue. In this scenario, since a send that is intended to be lost is still on the queue data structure, the next reception would match this old send—thereby throwing all the send-receive matches out of sequence. To avoid this, the transmission should be explicitly marked as a loss. Figure 22 shows a lost message transmission for the first packet (data(0)) sent.

Messages can appear to overtake each other when being transmitted between two modules. This would seem to defy reason as the modules are considered to be separated by a

single queue. This works because a TSD diagram abstracts away the name of the queue used by not explicitly *naming* the used queue. In this way a second queue can be used to connect two modules and by changing the order of the respective receptions the TSD can appear to show a reordering of messages. Figure 23 provides an example of this.

This non-FIFO effect is restricted to the generic format because PROMELA channels are queues and queue contents can not be reordered. It is possible that an abstraction of a network that can reorder messages could be formulated by manipulating the ***tsd** input, as is seen for the generic format, but so far PROMELA output only models "point to point" data transmissions.

6.3.4 Discussion

The **tsd** tool is particularly useful when trying to understand or debug a PROMELA model. A random simulation of the model gives a quick view of how messages travel. If any obvious problems are seen the model can be changed. If no obvious errors are found, the model can then be verified. If an error trail is produced, a simulation is rerun with the trace option. This hopefully highlights where problems lie so that they may be corrected. If the model has been verified, another random simulation will give a clear indication of how the model "normally" works. Each time the model is iteratively changed, we may have to follow this cycle.

***tsd** has been helpful in spotting design errors that passed the analyser's validation and verification tests. For example, we have created models that did pass the verifier tests but when we ran a random simulation found that the simulation did nothing but time out on both sides—in effect doing nothing, but by doing nothing, it did nothing wrong. A model that is not in error by virtue of doing nothing is not a correct design. We want to avoid 'trivially correct' models. When the tool is run as a random simulation, one can readily see if the message interaction 'makes sense'. These design problems are very easily spotted with ***tsd**.

6.4 tsd, the Tool

"***tsd**" works essentially as was explained above for the 'by hand' approach. It takes as input the result of a SPIN simulation with the **send** and **receive** event flags set. Its output

consists of a two dimensional graph, bounded on each side by an x axis and a y axis. On the x axis, each module that sends or receives is assigned a column. The y axis represents an event time line with each send or receive event representing a further 'clock tick'. Time increases down the y axis. Each message arrow will point at a downwards angle, connecting the sender to its receiver.

Each input line is queried for either a send or receive event. Send events are placed at the start of the respective channel or queue. Subsequent send events are placed at the end of that queue, if the channel is the same, or else at the beginning of a new queue. Any receive event will correspond to the send event stored at the head of its matching queue. At this point a match is found so a 'line' is drawn to show a completed **send** and **receive** and the head of the queue is deleted in order to remove the message from the queue. Receptions that are 'lost' due to 'lossy' channels are marked as such.

After the entire model is processed, the channels are checked for any messages that were sent but never delivered. These remaining messages may indicate that there are bugs in the design, but not always for example, if the synchronising handshake fails, the model will give up possibly with un-delivered messages still remaining in its channels.

6.4.1 Calling Syntax

The input to ***tsd** is the output of 'spin -s -r', which may be piped to ***tsd** or written to an intermediate file which is then read on ***tsd**'s standard input.

The program takes its data from the standard Unix input, stdin. Its usage can be illustrated as follows:

```
cat <filename> <filename>|xtsd <options> or
xtsd <options> < <filename>
```

but unless several 'generic' files are used, the first option is of extremely limited use because the output is often contained in a single file or is the standard output of the SPIN tool as the next example shows:

```
spin -t -s -r <Promela-model-filename> | xtsd &
```

We pipe the SPIN output directly to **xtsd**, which then extracts the appropriate information and then draws the TSD. The SPIN options indicate that we are examining the message

transmission of an error trace. The lack of options on **xtsd** indicate that the version of SPIN is version 2.0 or higher.

6.4.2 Layout

The assigning of the columns to the proctypes is done using an heuristic that assigns increasing column indices in the order of the instantiation and sending of a message over a channel. This may be over ridden by a saved layout file, "layout.tsd" in the current directory. In fact, once a good layout is decided upon, the layout file is exclusively used and the heuristic calculation is no longer necessary.

The layout file may be edited using any text editor. In this way, ***tsd** assists the user by using a heuristic to guess what the best proctype ordering should be but still allows the user to easily see and change this layout.

Some models use a single upper user module to test the underlying protocol. The layout heuristic can not guess this so the user must add this information manually to the layout file. The proctype id and the channel id are added to the layout file. The program knows that if a proctype id is seen more than once the next number is its channel id and not a new proctype id.

6.4.3 Main Program

The ***tsd** code processes input as follows:

1. check for first command line argument after **tsd**
   - **'g'** generic
   - **'p1'** SPIN version 1.0–1.6
   - **'p2'** SPIN version 2.00 and up, which is the default
2. set variable 'X'; 0 for ASCII mode, 1 for X11 windows mode
3. look for file in cwd that identifies a line number in the PROMELA code that corresponds to an action of receiving a message and then 'dropping it on the floor'. (setting LOSS_LINE).
4. include the function library

5. call the three main processing functions

xtsd

The X windows version has additional code to handle the interface, functions for such things as:

- window manager setup,
- setting up the window user input buttons, scroll bars, and the drawing area "canvas".
- "smart" horizontal and vertical scroll bars
- allow resizing of application

The main interaction with the program is through the four buttons: scale, quit, "print portrait mode in Postscript", and "print landscape mode in Postscript". The scale button allows the user to shrink the entire model graphic down to a more manageable size. See figure 24 to see what the X windows version, **xtsd**'s user interface looks like.

The PostScript output option produces excellent graphics that can be included in papers to illustrate the actions of the protocol. For example, **xtsd** can produce a drawing of the expected message interaction of messages in an error free environment in order to give the new user a sense of how the protocol should work under normal conditions.

The 'print Postscript file' function always gives the Postscript file a unique name based on Unix pid, mode (landscape or portrait) and number of times a print out has been made in a particular node. Actually the 'print' is a misnomer in that the file is saved to the current directory, where the user may then view it or print it.

6.4.4 TSD library program

Most of the functionality for **tsd** and **xtsd** is contained in a single library file, "tsd-lib.tcl". What follows is a short synopsis of these functions, in the order they are invoked:

1. look for 'layout.tsd' file and if not found open one up for writing; otherwise read in and assign proctype id's to x value columns and columns to proctype id's

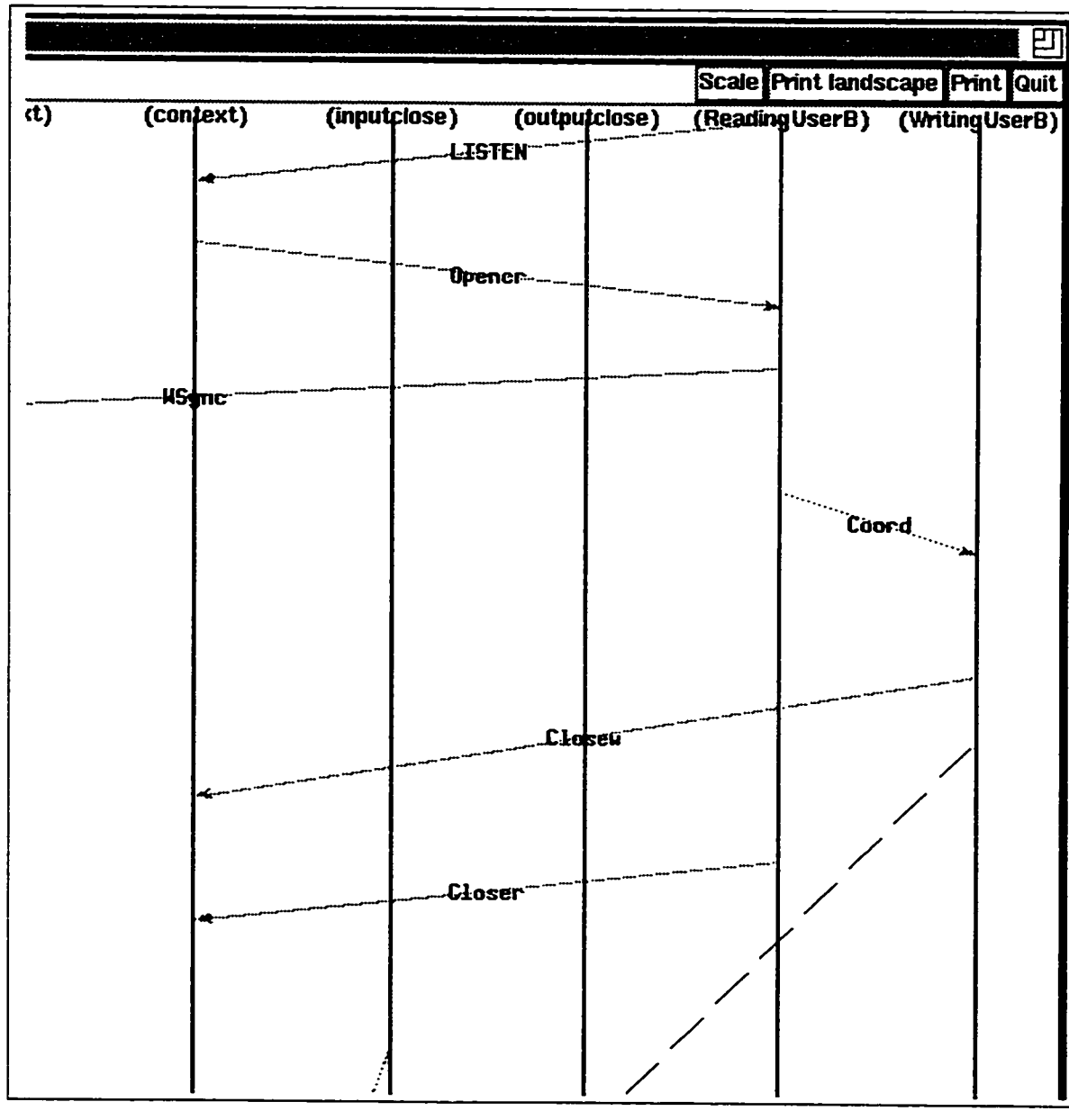


Figure 24: **xtsd** interface

2. decide on parser ('g', 'p1', or 'p2') and start reading input data line by line; or exit if can not determine type of input data file requested

3. create associative arrays from input line indexed by input fields

4. test for Send or Recv lines, ignore all the rest

   (a) if it is a Send line, add contents of same indexed associative arrays as a list to queue indexed queue data structure.

   (b) if it is a Recv line, remove top of appropriate queue that corresponds to Recv

      i. if !X then create 2 array entries indexed by occurrence of appearance in input data

      ii. if X then draw an arrow between the appropriate modules with message content label in middle
      use 'xwininfo' to determine X monitor's colour plane

         A. if it is black and white , draw arrows in stippled black to approximate gray

         B. if it is colour monitor use gray or the first field will determine the colour drawn on the screen (from verification work this is usually red, blue, or yellow)

5. if anything is left on the input queues after the input data file has been read in, then signify this as follows:

   (a) if X, put an error icon and the message contents in a large font at the position where its arrow tail would have been drawn if the message was completed

   (b) if !X, add to the array and append the string 'dead' in order to signal message loss when the entire array is printed

6. print out

   (a) if !X print out the entire array

   (b) if X draw the vertical lines and column headers

At the end of the library file, there are the other parsing functions for the generic and pre version 2 input formats and some X functions to scale the window and provide some intelligent vertical window scrolling. The window does not scroll past the area allotted for the TSD and the heading always stays at the top of the screen when the user is scrolling so as to always indicate which column refers to which proctype.

The last part to mention is the "obsolete" system wide header file.

6.4.5 The "tsd.h' File

A certain class of models are written in which the network module is subsumed into a higher level module. Message loss is 'simulated' by the receiving module not passing on the message or "dropping it on the floor". Message loss is mapped to the line in the source code that receives the message at the receiving module but then does nothing with it. The user then sets the LOSS_LINE variable with the appropriate line number and any receptions that are received at that line number will be visually flagged as a message loss.

The tsd.h file contains this single variable assignment and initially was a system wide file but was moved to the current directory of the model under examination. As the model changes the specific 'loss line' changes as well, so the user is reminded, at first, to remember to change this line in the current directory. If this class of model is not being used, the user can ignore this warning.

6.4.6 Bug

Proctype to column assignment may be inaccurate under the following condition: when a proctype ends and a new one is instantiated right away, the new proctype takes the old pid. Since our program only checks for 'proctype name' and 'proctype id' to map it into a column it appears that the first proctype never finishes and that the second proctype is a continuation of the first. This was unexpected, as we had thought that any new proctype instantiation would receive a *new* proctype id number.

In order to correct this problem, we will have to check other criteria when trying to uniquely map a proctype into a column—perhaps the list of channels assigned to a proctype can help to identify the proctype column mapping or the line and state number from whence the proctype was instantiated (or "run") in the PROMELA source code. Fortunately

this error is very rare and easily spotted.

Chapter 7

Drawing Finite State Machines

7.1 Introduction

We have created a suite of tools (see table 7) which we use to draw finite state machines (FSMs). FSMs are used to describe the inner workings of a module, see chapter 3. Most often the module we examine is a component proctype of a PROMELA system model. We could also use these tools to draw the global FSM for the entire system model but because of the 'state explosion problem', this is often unfeasible. Finally, these tools also allow us to construct our own FSM description entirely independent of a PROMELA specification.

7.2 Background

FSMs have been a staple of protocol designers, specifically in the specification and validation phases. FSMs document the internal activities of a system model by describing the actions at the proctype or module level.

Until the second version of SPIN was released, the only FSM representation available to PROMELA designers was the line-oriented output from "pan -d" (see figure 25). The figure shows state transitions for the single proctype 'context'. Each line represents a transition from a start to an end state. Also including is the line number and the code from the PROMELA source file, as well as other information. There is, of course, a representation for each and every proctype in the system model. We use this output as the input to our drawing tools.

We have already described the limitations of a non-graphical representation. In this

```
proctype context
        state  50 -(tr   9)-> state  15  [id 205 tp 523] [----G] line 8 => fromW?Openw
        state  50 -(tr  29)-> state  31  [id 221 tp 525] [----G] line 8 => fromR?LISTEN
        state  50 -(tr  42)-> state  50  [id 237 tp 526] [----G] line 8 => chin?RCV_WCLOSE[n \\
],RCV_RCLOSE[n],RCV_END[n]
        state  50 -(tr  43)-> state  36  [id 239 tp 523] [A---G] line 8 => fromW?Closew
        state  50 -(tr  45)-> state  39  [id 242 tp 525] [A---G] line 8 => fromR?Closer
        state  50 -(tr  47)-> state  42  [id 245 tp 523] [A---G] line 8 => fromW?Eofile
        state  50 -(tr  49)-> state  45  [id 248 tp 525] [A---G] line 8 => fromR?Eofile
        state  50 -(tr  51)-> state  48  [id 251 tp   2] [A---G] line 8 => (((finish_i[n]==1

Transition Type: A=atomic; D=d_step; L=local; G=global
Source-State Labels: p=progress; e=end; a=accept;
```

Figure 25: Sample "pan -d"Output (spin 2.3.3)

regard, New's work on FSM graphics has been very influential. (As of SPIN, version 2, XSPIN also supports a graphical representation for FSMs.)

Thus, the problem we examine in this chapter is how to draw out a FSM. We have found that the problem can be relegated to two distinct phases. The first phase revolves around choosing an appropriate two dimensional spacing for the FSM. The second phase simply draws the FSM based on the spacing.

There are benefits of being able to draw out a FSM. Most important to us, we want to draw out a FSM so that the designer can 'visualize' and possibly adjust the design, until a satisfactory design can be finalized. As for all visual representations, a figure aids in our comprehension of the system we are examining.

7.3 The FSM Drawing Tools

Our system consists of several programs in which are meant to be run as filters or strung together in a pipeline where the functionality of the 'tool' is distributed over these relatively small programs. Table 7 contains a list and short discription of the tools used to draw a FSM. Appendix C lists the source code for these tools. This reflects our preference for small tools that can be mixed together to provide larger more powerful tools—the classical 'Unix style tools'. This style is also refelective of their development: small tools can be experimented

List of FSM Tools	
Name	**Action**
fsm-ls	list all proctypes in the 'pan' file
fsm-make	read specified proctype's FSM from output of pan -d
fsm-atomic	symbolic link to 'fsm-make' (for 'compacted' FSM)
pre-xdag	process FSM with better 2 dimensional layout algorithm
line-only	strip out 'state value' from node
empty-node	strip out all information from node
xdag	draw a graph
global-fsm	create a global FSM of the entire model

Table 7: List of FSM Tools

with to test the feasibility of certain concepts before we commit to a specific one. Eventually, the tools could be very easily integrated into a single window based package[1].

7.3.1 Smaller Tools

In this section, we will provide a discussion of the smaller tools and on how all the tools are used together to deliver a user's requested FSM diagrams. The major tools will be discussed in their own sections.

fsm-ls

The first tool 'fsm-ls' is used to present a list of all the possible proctypes in the model. In this way a user can then decide on which proctype should have an FSM drawn for it. Note that each proctype must be explicitly chosen when a FSM is drawn for it, this is not done automatically for each existing proctype.

The script uses the 'pan' file with the '-d' option as input. Figure 25 shows an excerpt from the output that 'pan -d' produces. All lines with a 'proctype' in it are pulled out and the second field, its name is then printed.

[1]The tools are written in Tcl/Tk, awk and C as they were (and for similiar reasons) for *tsd. The graphical interface is for the X windows environment. Unlike the *tsd tool, it is impossible to draw an FSM in an ASCII format. Instead, we have opted to write out ASCII format in a compact representation to the standard output. This file can then be used either as input to other programs or to examine the details of the FSM connectivity on an ASCII terminal.

We use the 'pan -d' out as input because if a PROMELA source file is used as input as in XSPIN, the '#included' files are not listed. This could be changed by including the files into the main buffer whenever an '#include' is read in the text, however, the line positions from the original source would no longer be valid. Thus we use the output of 'pan -d' as it lists all the proctypes and maintains the correct line number for each file name in the source code. The list is returned to standard output (but could just as easily be written to a list widget).

fsm-atomic

This tool is not a 'file' in the normal sense, but a 'soft link' to the file 'fsm-make'. Unix allows us to create a 'link' (like a pointer) to another file. The advantage of this is that we can ask from within the Tcl code, what name was the executable invoked by (similar to other Unix shell scripting languages and the programming language 'C') and then act upon this information. If the file was invoked as 'fsm-atomic' the file fsm-make attempts to compact or minimise the FSM by eliminating 'unnecessary' nodes in the graph of the FSM. We will discuss these compaction issues shortly.

"line-only" and "empty-node"

"line-only" and "empty-node" are two filters which provide similar functionality—they modify the information stored in each state node label. The former reports only the line number as the label and the last strips out the label completely.

They are usually used in a command pipe line where they filter the input to the xdag drawing tool.

global-fsm

This program is used when the user wishes to produce the global FSM of the entire model. Its use is not recommended unless the global FSM is particularly small. "global-fsm" is a pipeline of Unix commands consisting of the tools 'dag.awk' and 'dag.tcl'.

'dag.awk' converts global FSM information from SPIN into 'pic' format, which is a System V Unix tool which we do not have access to. 'dag.tcl' converts 'dag.awk' output into 'pre-xdag' format. This output is then ready to be used with the rest of our drawing tools.

Overview

Figure 26 should help to put this discussion into perspective. It serves as a roadmap so that following any path from the 'root' to a 'leaf' will exhaust all the ways our tool can be used. 'fsm-ls' is used by the user to determine which proctype to draw. Then either 'fsm-make' or 'fsm-atomic' is used to calculate the coordinates for either an uncompacted or compacted graph. At this point the output is ready to be drawn with 'xdag' or is passed to 'pre-xdag' to have an alternate layout calculated. After 'pre-xdag' is finished the output is in the format appropriate for the 'xdag' tool. The user may choose to send it directly out to this drawing tool or pass it through either the 'line-only' or 'empty-node' filter and then to 'xdag'.

7.3.2 fsm-make

This program is the first tool used in the path from reading a PROMELA model to producing a specific proctype FSM. "fsm-make" reads the output of the 'pan -d' file (figure 25) and extracts all the information required to draw a FSM for a specified proctype.

Command Line Options

The program can be run with several options (the first, 'fsm-atomic' has already been mentioned). The program can output the proctype information in the input format for the drawing tool, 'xdag' (figure 33) or it can create the format for the 'pre-xdag' tool (figure 30). As well, the user can request that the actions (the actual PROMELA code) from the PROMELA model for each state transition in the system be included in the graphic. See below for a list of these command line options, listed by index order:

0 name of file that invoked program, either fsm-make or fsm-atomic

fsm-make	reports full FSM as reflected in pan -d
fsm-atomic	reports compacted FSM , with 'extra' states removed

1 FORMAT either x or pre, default = x

x	for output in xdag compatible format (output is ready to draw)
pre	for output in pre-xdag compatible format ('ASCII' FSM)

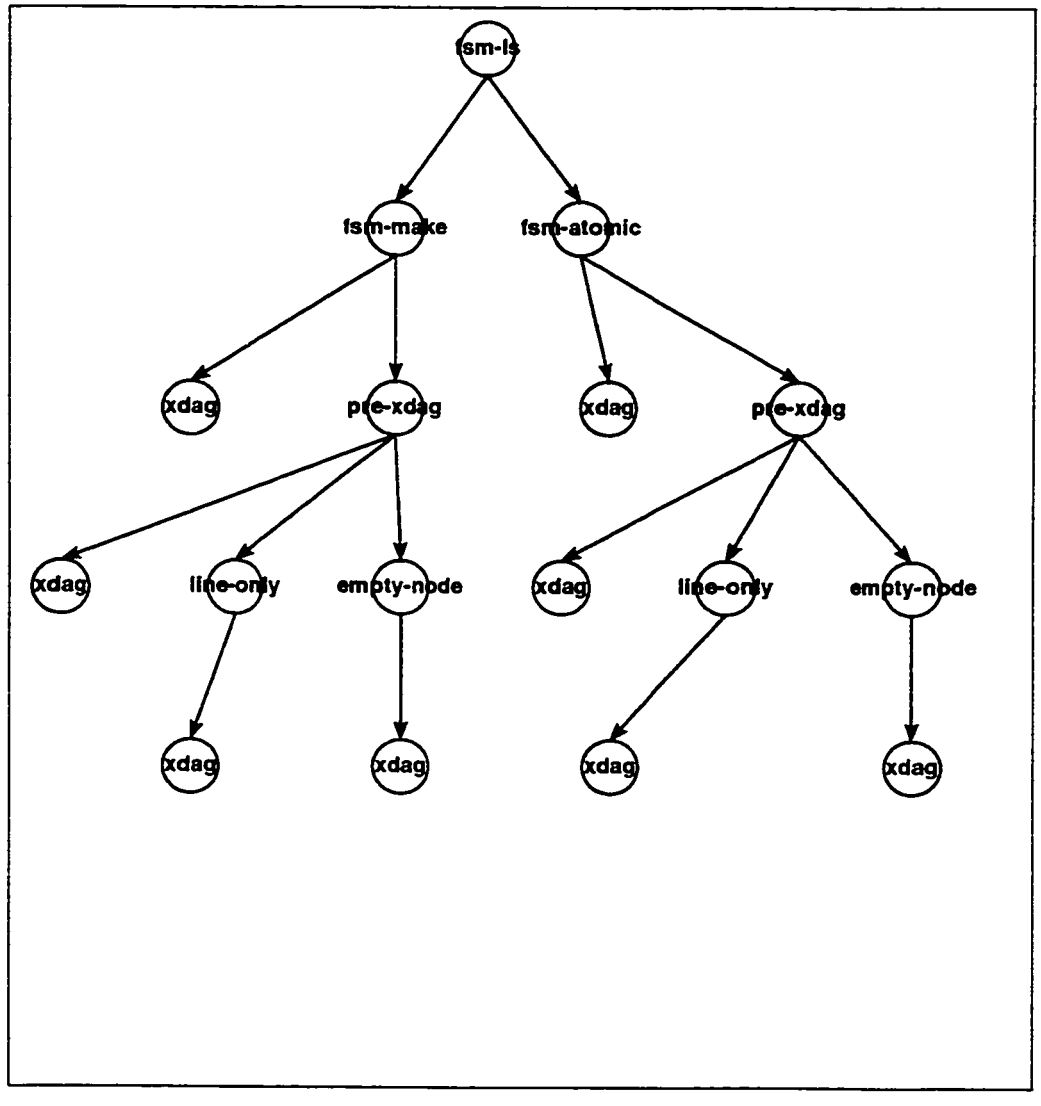


Figure 26: FSM Design Methodology Overview

2. TLABEL either tu or tl, default = tu
   - **tu** for transitions, unlabelled
   - **tl** for transitions labelled

The user must give the command line arguments in the order listed above. If later arguments are left out they assume the default values 'x' and 'tu' (for command line options 1 and 2).

Command line options 1 and 2 are assigned to the internal variables, FORMAT and TLABEL, respectively. We will discuss these format structures when describing the tools that read them.

In order to clarify these options, we illustrate their effects by presenting the rendered graphics for a sufficiently complex FSM based on an XTP Context Machine model which we wrote in PROMELA. The Context Machine module manages the various connections or contexts in an XTP machine. See figures 27–29, figure 31 and figure 32.

There are two FORMAT choices so that two drawing algorithms can be accommodated. The 'x' option chooses the algorithm that Holzmann uses in XSPIN. It is fast and simple but the drawings are 'ugly'. Figure 27 shows the result of applying this algorithm.

The 'pre' option writes out the FSM in a format for another drawing algorithm (pre-xdag format) which is more complicated but produces much more satisfying drawings. Figure 28 shows the resulting graph that the 'pre-xdag' program calculated.

The pre-xdag format is seen in figure 30. This is used as input to the 'pre-xdag' tool which then produces the format as seen in figure 33.

Both figures 27 and 28 were drawn from the same initial 'pan -d' output.

Compacted FSMs

When a section of code is marked as atomic in a PROMELA model, the code is executed as a single unit and can not be interrupted until it is finished. Conceptually, the component actions can be abstracted away.

Likewise, the compaction option looks for all atomic sequences and tries to compact all its linear sequences, as these are considered extraneous or redundant. The output consists of a minimised FSM wherein the basic framework for choices is all that remains.

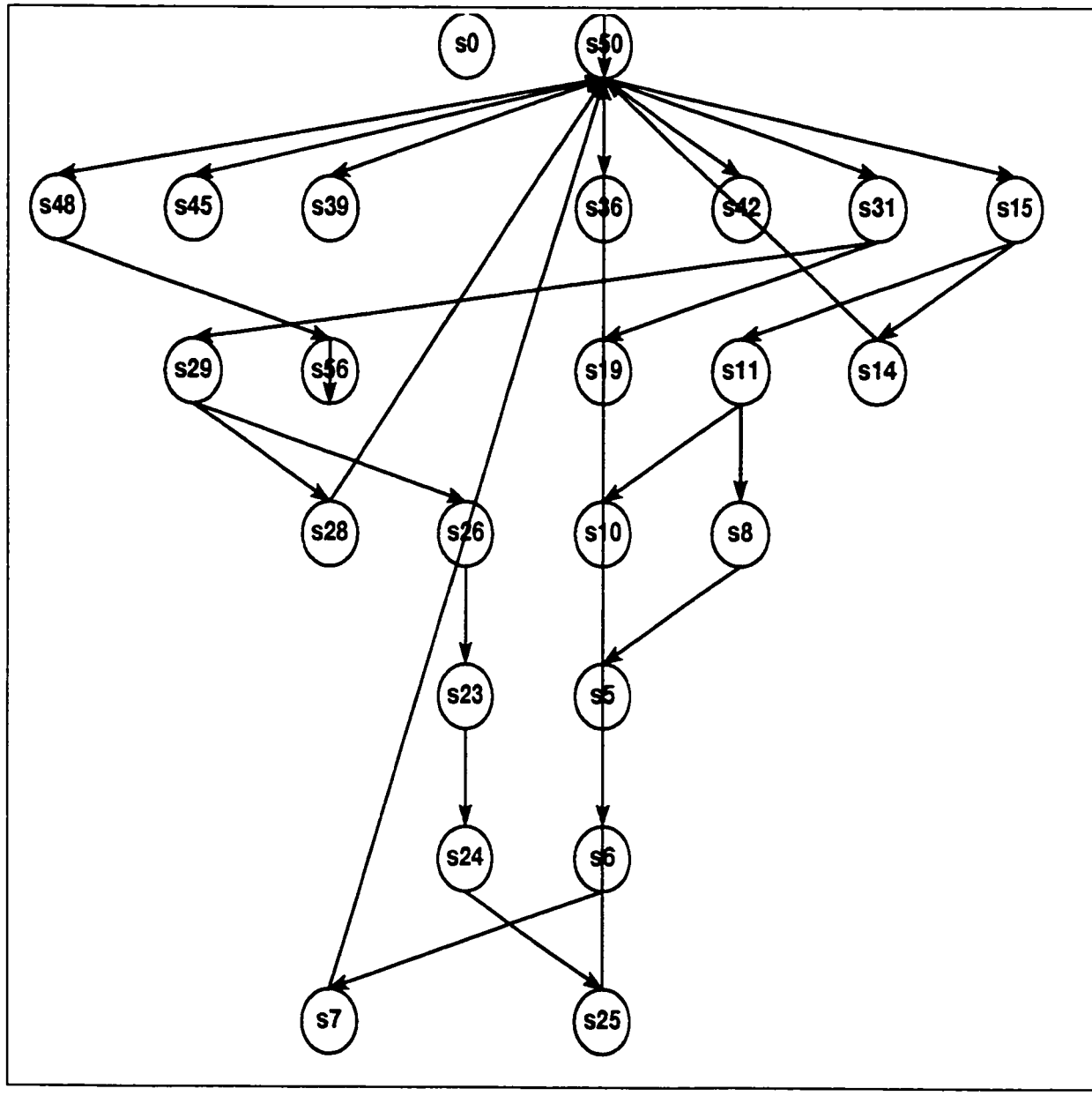


Figure 27: Context Machine (Holzmann drawing algorithm)

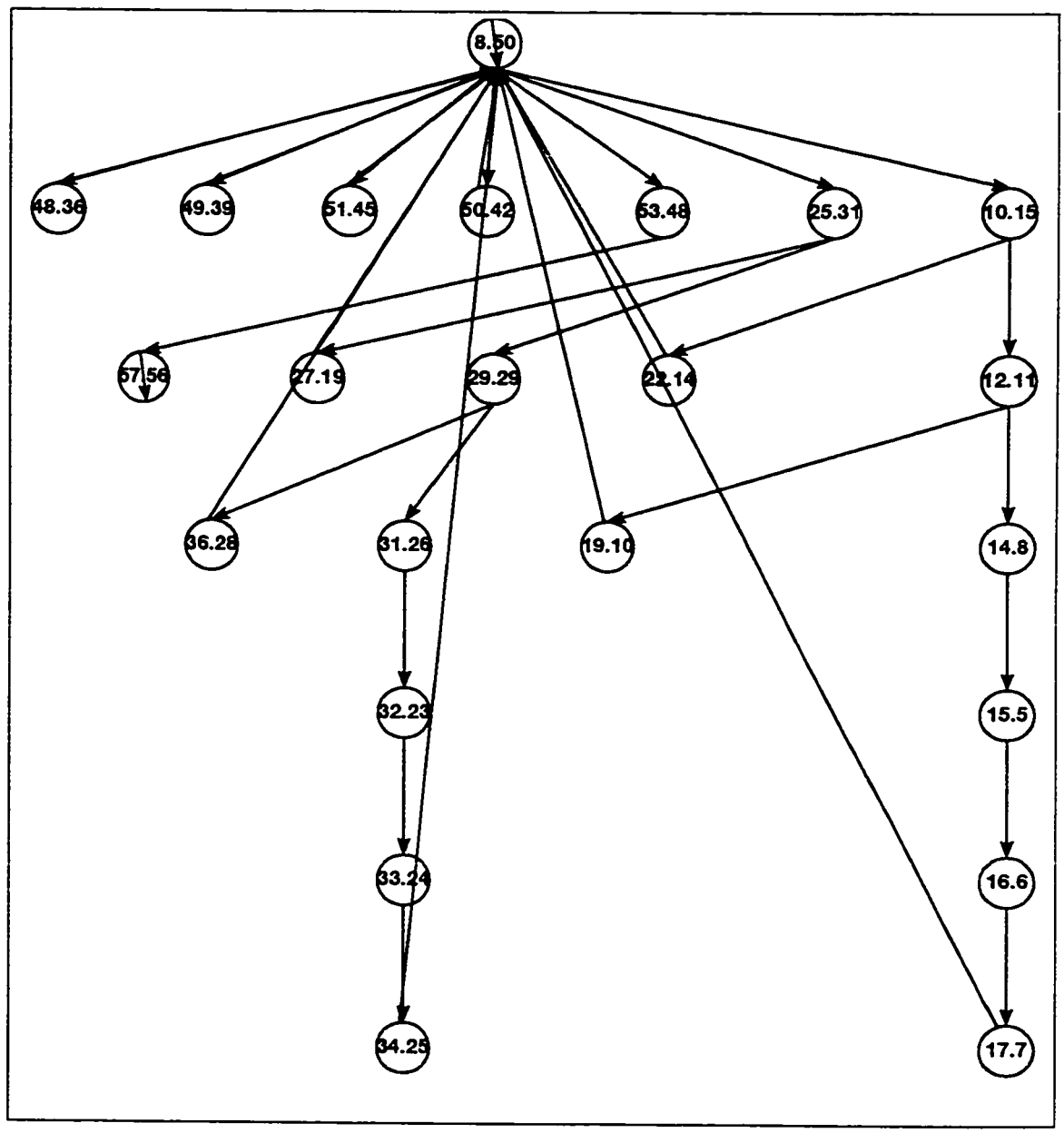


Figure 28: Context Machine

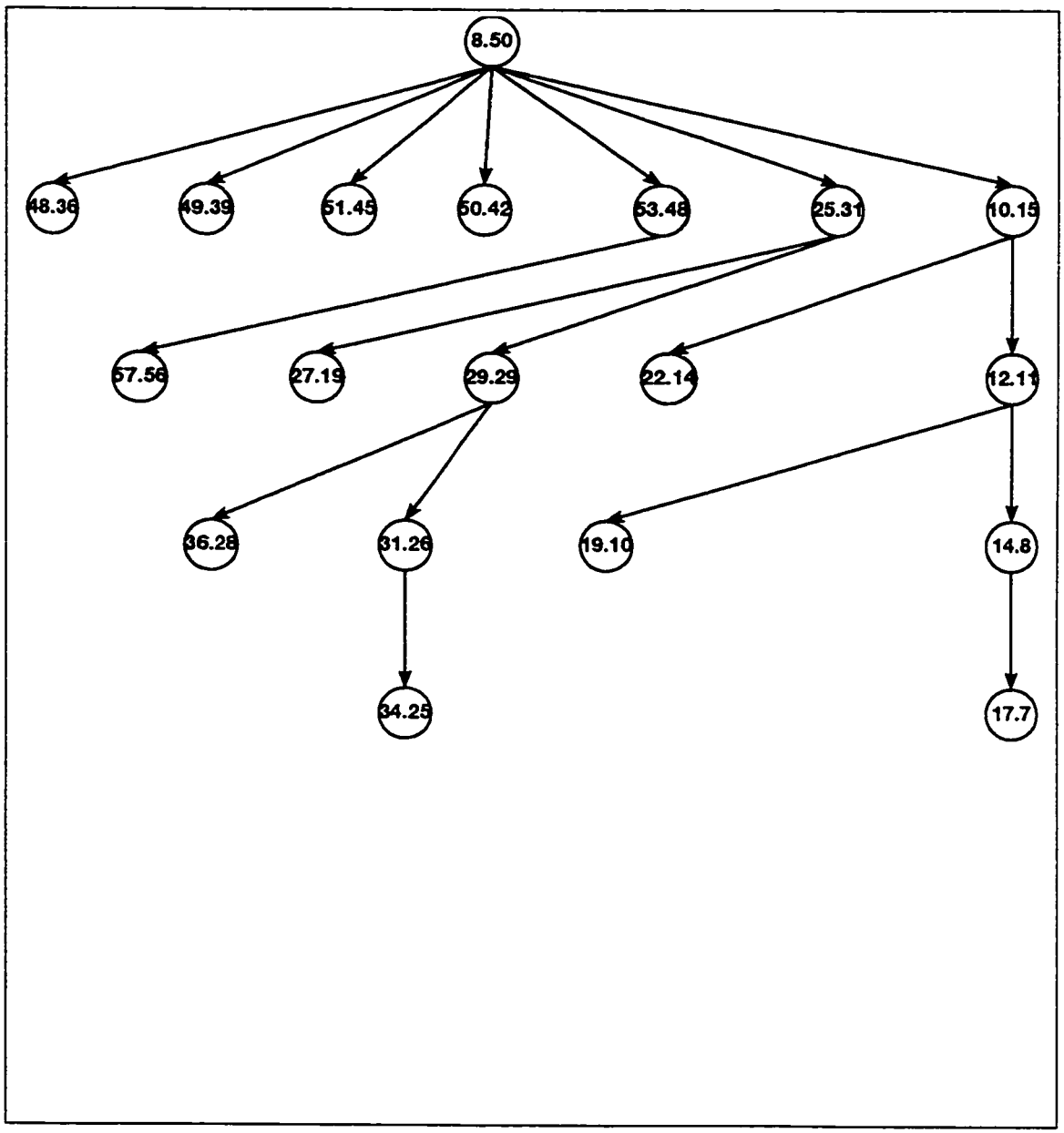


Figure 29: Context Machine (compacted "down" transitions)

In figure 29, we show the result of compacting the graph from figure 28 by using "fsm-atomic" instead of "fsm-make". For clarity we show only the "down arrows" (see the discussion in 'pre-xdag').

We have investigated the possibility of having the tool eliminate all but the start and ending states involved in an atomic sequence but feel that the previous approach is better, as it is important to know the structure of a model, even for atomic sequences, and so we felt that we would not want to eliminate this kind of information.

The fsm-atomic option is used when we wish to compact certain atomic sequences but we re-emphasise that the concepts of compact and atomic are not identical. There are some atomic sequences where one would want to see all the component actions—even for linear sequences. If the user wishes to eliminate sections of the FSM, for abstraction purposes, this can be done be eliminating the nodes from an intermediate FSM input files. Once identified, the user can eliminate any nodes, including the atomic ones.

An ASCII FSM

The output created when using the 'pre' option also serves as the basis for generating an ASCII representation of the FSM. This can be accomplished by typing:

```
fsm-make <proctype-name> pre |sort -n
```

at the Unix command line prompt. The pipe to the Unix 'sort' command is necessary in order to find the starting node. The '-n' requests a numeric sort, node labels are based on source code line number and SPIN assigned state number, both numeric values so this option should not be left out. We could have added this functionality to the program but we did not see the need. It probably will not be used too often and the functionality is already in the system.

It is important to realize that the lowest valued node *is* the *first* or *start* node. Consequently, we could record the lowest node value or even the first node read. The reason that these lines are not kept in order is due to how Tcl stores the order of associative arrays (essentially randomly as the index is based on a hash value).

7.3.3 Discussion of the code for "fsm-make"

The "fsm-make" algorithm is as follows:

1. check to see if the analyser 'pan' exists and if it does not tells the user to compile the model with spin -a to produce the analyser, 'pan'

2. read output of 'pan -d', and determine if the current input line represents a proctype or state transition

3. if a "proctype line" is found, check if it is the requested proctype and if it is then keep reading "state transition" lines until the next "proctype line" is found

4. parse off "source state", "destination state", "line number of source state" and "action (or PROMELA code) of source state" from the fields of the input lines and calculate distance the between source and destination nodes (levels), all the while tracking the maximal level reached as you construct the graph for the FSM

5. if the file was invoked as 'fsm-atomic' then re-parse 'pan -d' and record the atomicity of each transition and for each current node, calculate its parents

   (a) look for atomic segments to compact (I did not include the d_step keyword). There are four cases as the current node's transition can either be atomic or non-atomic and the same is true for its parent (the parent that leads directly to the current node) If the current node is non-atomic do not change the graph. If the incoming transition was non-atomic and the outgoing transition is atomic do not change the graph but mark this as a transition to an atomic section by recording the current node as the 'ancestor node'. If both the incoming and outgoing transitions are atomic, this is a potential path to compact wherein the ancestor's child is replaced with current target node. However this can only occur if the ancestry is linear.

6. if the 'pre' option is selected, print out the FSM in the format for pre-xdag

```
src:trg,trg;
```

or

```
src:trg tlabel,trg tlabel;
```

```
14.8:17.7;
31.26:34.25;
12.11:14.8,19.10;
29.29:31.26,36.28;
53.48:57.56;
25.31:27.19,29.29;
8.50:10.15,25.31,48.36,49.39,50.42,51.45,53.48;
10.15:12.11,22.14;
```

Figure 30: "pre-xdag" Input Format

otherwise pass the FSM to Holzmann's two dimensional layout calculation algorithm. This algorithm calculates the positions for the node and transitions for 'xdag' tool to draw.

7.3.4 pre-xdag

The program 'pre-xdag' is based on a Motif public domain program, 'xdag' that is used to draw directed acyclic graphs. It was adapted to calculate the two dimensional layout of the nodes and transitions for proctype FSMs. Its algorithm produces results superior to those produced by the other (Holzmann) drawing algorithm.

'pre-xdag' can not handle looping cycles in its graph, so all cycles must be removed such that only 'down' transitions are included, then these node positions are calculated. The 'up' transitions are later calculated by referring to the coordinates produced for the graph with only the 'down' transitions included.

In figure 31 we illustrate the "up" transitions and in figure 32 we show the "down" transitions.

We see a need to automate the following aspects of the program as soon as possible:

- need program to eliminate 'up' transitions
- program to parse out 'up' transition x and y coordinates

Figure 30 is an example of the pre-xdag input format. By sorting these lines we get the ASCII FSM. We can see quite easily that the row that starts with "8.50" contains the start node for the graph.

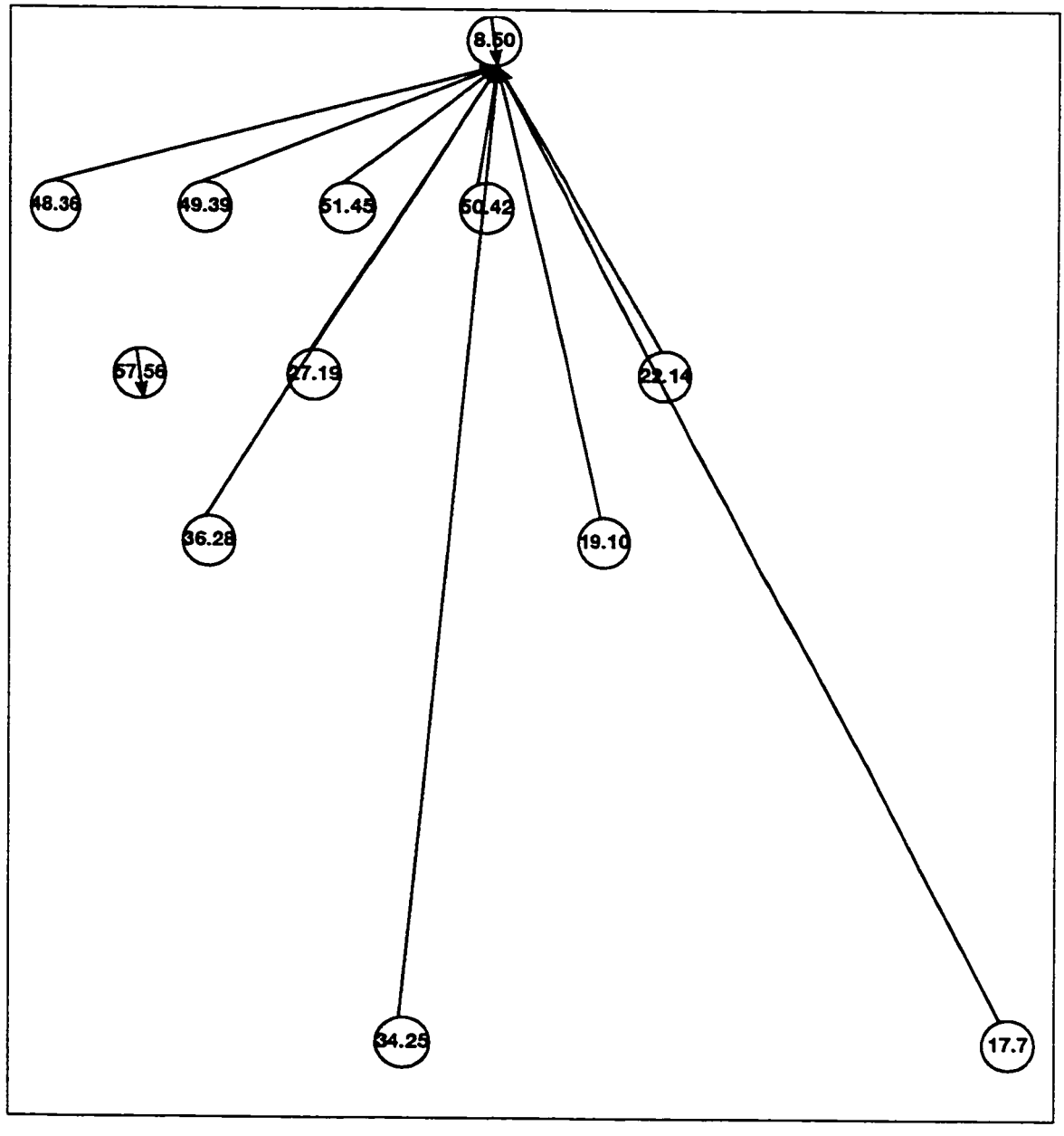


Figure 31: Context Machine ("up" transitions)

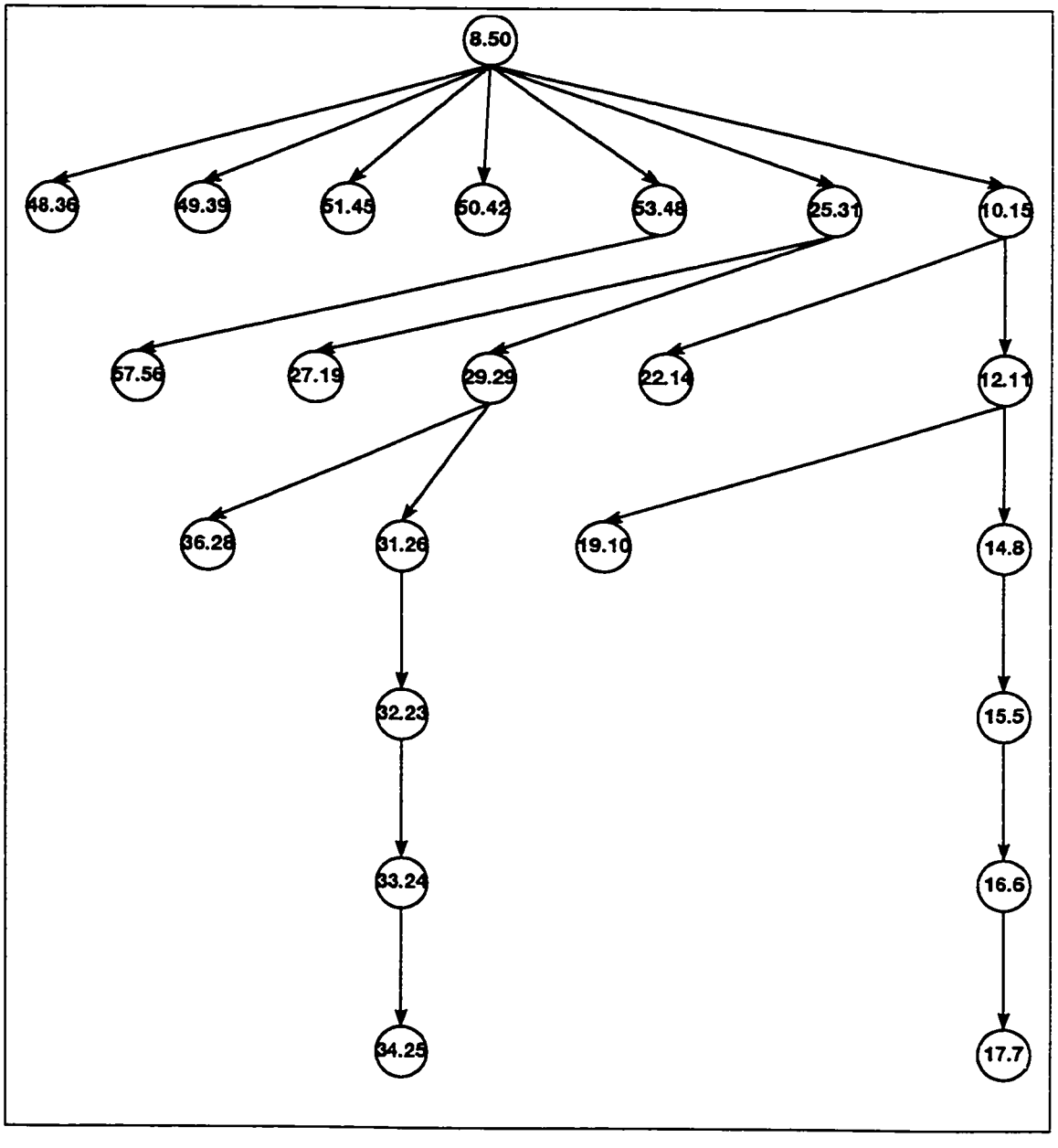


Figure 32: Context Machine ("down" transitions)

```
8.50 400 15 7 415 45 711 115 415 45 611 115 415 45 165 115 415 45 252 115 \\
415 45 411 115 415 45 333 115 415 45 511 115
48.36 150 115 0
49.39 237 115 0
51.45 318 115 0
50.42 396 115 0
53.48 496 115 1 511 145 215 215
25.31 596 115 2 611 145 315 215 611 145 415 215
10.15 696 115 2 711 145 711 215 711 145 515 215
57.56 200 215 0
27.19 300 215 0
29.29 400 215 2 415 245 365 315 415 245 255 315
22.14 500 215 0
12.11 696 215 2 711 245 711 315 711 245 480 315
36.28 240 315 0
31.26 350 315 1 365 345 365 415
19.10 465 315 0
14.8 696 315 1 711 345 711 415
34.25 350 415 0
17.7 696 415 0
```

Figure 33: "xdag" Input Format

7.3.5 xdag

Although this is the program that draws the FSM, it knows nothing about graph structure. Its job is to draw circles and a variable number of arrows that leave from the circumference of the circle. It reads a data file where its input lines can be entered in any order, such as figure 33. Labels can be drawn in the circles and across the arrows at the user's discretion.

The modularity of this design allows us to reuse this program to draw anything that can be represented as a set of circles with outgoing edges or arrows. The user imbues meaning to the interpretation of circles and arrows in any way that they please but here we interprete the graph as an FSM by designing the input layout and objects to be used such that circles are used for nodes and arrows for transitions. The coordinates for the nodes and arrows are pre-calculated, so that 'meaning' is already decided upon by the time 'xdag' reads it.

Figure 33 shows an example of the xdag input format. The line with the '\\' at the end of it should be a single line that continues with the one immediately following it. The line was broken for readability.

The data input file comes from either fsm-make or pre-xdag and consists of a series of

formatted lines, structured as follows: first the node label and its positional coordinates, followed by the number of transitions that lead from this node, next there are a variable number of coordinates (possibly zero) for each transition. This is expressed in terms of positional coordinates for the tail and head of the arrow. After each set of "arrow" coordinates, if a non-numeric field is found it is interpreted as a "transition label" that is drawn over the midpoint of the arrow otherwise the next set of arrow information is read. This is done automatically for each arrow.

The xdag input format is as follows:

- node label
- x coordinate for top left circle
- y coordinate for top left circle
- x coordinate for bottom right circle
- y coordinate for bottom right circle
- number of transition from node,
- variable number of "arrow information" entries

The arrow information entries are contingent on the number of transitions. If there are no transition then there will be no arrow coordinates, otherwise there is a set of coordinates per transition in the following format. For each arrow, there may be a label.

- x coordinate for arrow tail
- y coordinate for arrow tail
- x coordinate for arrow tip
- y coordinate for arrow tip
- possible transition label (must be non-numeric)

7.3.6 dag-lib.tcl

The code for 'xdag' is written in two files, 'xdag' and 'dag-lib.tcl'. The file 'xdag' sets up the X window parameters, see the discussion of 'xtsd'—the files are virtually identical. The main functions for 'xdag' are contained in the library file 'dag-lib.tcl', where the functions to draw out the nodes and transition arrows, and where the checks if there are any labels on the arrows occur.

7.4 Discussion

The two-dimension position calculation for graph layout is an important and necessary step. It is easy enough to derive the connectivity of the graph components and easy enough to draw a graph once the coordinates have been calculated, but *calculating* the points required to draw out the graph is difficult. One reason is because the node placement is an aesthetic decision and computers are not as good as people are at making this kind of decision. It can be argued that computers *can be made* to be good judges of aesthetic content by applying the techniques of artificial intelligence and expert systems, however, ultimately the set of criteria would have come from a human and furthermore since humans will be looking at the graph layout, a person's aesthetic criteria should win out. As well, different people may not agree on the same aesthetics. Therefore a computer must help the person and should use an heuristic that gives the best possible results, but still allow the person to over rule it. The computer should do "most" of the work, as the task can be overwhelming, but once the user "has something to work with", the user should be allowed to do so.

If we compare how graph layout is calculated in GROPE, the reader may be surprised to find that no calculations are done. The graph connectivity is calculated and drawn and then the nodes and their connecting transitions are literally dumped out onto the screen. The program expects that the user will move the nodes around in a satisfactory way. This layout can then be stored in a file, which is then used to calculate the layout of the graph.

The code could be improved as follows:

- allow mouse movements to rearrange the graph components (nodes and transition arrows)
- save the layout to a file

- (possibly) integrate all the programs into a single file.

The layout can be stored in a file, but it can not be easily manipulated since the file is often quite large (this is not the case for the '*tsd' tool which generally has a one line layout file). Being able to manipulate these components interactively on the drawing canvas would make the tool much more user-friendly. (Although mouse movements can rearrange the graphic, XSPIN does not save this layout to a file.)

Finally, and as is done for '*tsd', if there exists a layout that the user likes already stored in a layout file, the program should not recalculate a new one and should not write over the old one, but it should simply read it (since all that xdag does is draw, we would have to add functionality for this to that file as all the other files are involved in finding and then calculating the graph layout points in the graph).

Chapter 8

Conclusions and Future Work

8.1 Conclusion

Describing protocols in a precise, yet understandable way is a very difficult task. Informal techniques were initially tried but their lack of precision quickly made them insufficient for the increasingly more complex protocols being introduced today. As the protocol development community turned towards formal methods, imprecision was traded for understandability, as many of the description techniques are very hard to understand. In this thesis we have reintegrated the successful components of informal methods with the precision of formal methods.

We believe that generating figures from a formal model helps in the design process since humans are the ones that ultimately have to implement a protocol and humans benefit most from graphic representations. By grounding the figures on a formal model we reintroduce the very real benefits of clarity by relying on our inherent human design skills, without losing the mathematical precision and clarity of formal methods.

We have developed two tools: one for drawing the internal activities of the component modules of a protocol system, as expressed by the finite state machine model and one for drawing the external activities for the component system modules, based on the time sequences diagram model.

8.2 Future Work

This thesis can be expanded in several ways.

We have laid the foundation for the formal descriptions of a large, under specified protocol, XTP. We can see opportunity to expand on these models, to either make them more comprehensive in what they cover or to integrate the different models into a single, but much larger one. Formal models should also help to tighten up the specification of the protocol. Once this is done, only then should a natural description and figure description be produced—one that is based on the formal description.

We can see more research into the sophistication of the capabilities of the tools, specifically of the user interface and of the functionality. We would also like to see more layout algorithms added to the tools.

We could see adaption of the tools for other languages. For example, as we mentioned, it would be extremely easy to adapt the FSM tool for Valira. The TSD tool is also easily extended to include any other communication language or technique that uses message communication.

In future versions, we would allow mouse movements to rearrange and then save new layouts like the GROPE package (XSPIN can use the mouse to rearrange layouts but can not save them.). We also wish to write code to draw the overall model architecture for all the modules and their inter-connecting channels.

Finally, we would like to find out if there is a "universal representation", specifically a graphic one which could provide some sort of commonality to the different languages and techniques that are used to describe protocols in the protocol design community.

Bibliography

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, USA, 1977.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–60, 1987.

[BBBC93] Gordon Blair, Lynne Blair, Howard Bowman, and Amanda Chetwynd. Formal Support for the Specification and construction of distributed multimedia systems (the Tempo project) - final project deliverable. internal report MPG-93-23, Lancaster University, Lancaster, England, December 1993.

[BD87] S. Budkowski and P. Dembinski. An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[BH89] F. Belina and D. Hogrefe. The CCITT Specification and Description Language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341, 1989.

[BL91] Rezki Boumezbeur and Luigi Logrippo. Specification and Validation of Telephone Systems in LOTOS. University of Ottawa, May 1991.

[BL93] Rezki Boumezbeur and Luigi Logrippo. Specifying Telephone Systems in LOTOS. *IEEE Communications Magazine*, pages 38–45, August 1993.

[BNT94] T. Bolognesi, E. Najm, and P.A.J. Tilanus. G-LOTOS: a graphical language for concurrent systems. *Computer Networks and ISDN Systems*, 26:1101–1127, 1994.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Comm. of the ACM*, 12(5):260–261, May 1969.

[CA90a] S. Chamberlain and P. D. Amer. *A New User's Experiences and Impressions with Estelle*, pages 471–474. North Holland, Amsterdam, 1990.

[CA90b] Y.T. Cheung and J. William Atwood. Specifying the Xpress Transfer Protocol Using Estelle and Valira. In Jose Manas Quemada and Enrique Vazquez, editors, *Formal description techniques, III*, pages 503–517, Madrid, Spain, 1990. IFIP, Elsevier Science B.V. (North-Holland).

[DAC+89] Diaz, Ansart, Courtiat, Azema, and Chari, editors. *The Formal Description Technique Estelle*. North-Holland, Amsterdam, 1989.

[DB92] Amand Drayton, Lynne Chetwynd and Gordon Blair. Introduction to LOTOS through a worked example. *Computer Communications*, 15(2):129–34, March 1992.

[Dij75] Edsger W. Dijkstra. Guarded Commands, nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[DZ83] John D. Day and Hubert Zimmmermann. the OSI Reference Model. *Proceedings of the IEEE*, 71(12):1334–1340, December 1983. (special issue on Open Systems Interconnection (OSI) – new international standards architecture and protocols for distributed information systems).

[FA94] Ove Færgemand and Olsen Anders. Introduction to SDL-92. *Computer Networks and ISDN Systems*, 26:1143–1167, 1994.

[FLS90a] Mohammed Faci, Luigi Logrippo, and Bernard Stepien. Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach. University of Ottawa, Department of Computer Science, TR-93-07, 1990.

[FLS90b] Mohammed Faci, Luigi Logrippo, and Bernard Stepien. Formal Specifications of Telephone Systems in LOTOS. [LPU90], pages 25–34.

[FLS91] Mohammed Faci, Luigi Logrippo, and Bernard Stepien. Formal Specifications of Telephone Systems in LOTOS. *Computer Networks and ISDN Systems*, pages 53–67, 1991.

[GHP92] P. Godefroid, G.J. Holzmann, and D. Pirottin. State space caching revisited. In G.v. Bochmann and D.K. Probst, editors, *Computer Aided Verification IV*, pages 178–91, Montreal, Que., Canada, 1992. Computer Aided Verification. Fourth International Workshop, Springer-Verlag.

[Gla94] Bradford Glade. Temporal Logic to Never Claim Converter. Technical report, Cornell Universtity, Department of Electrical Engineering, Ithaca, New York, October 1994.

[Got92] Reinhard Gotzhein. Temporal logic and applications - a tutorial. *Computer Networks and ISDN Systems*, 24:203–218, 1992.

[GR92a] Paweł Gburzyński and Piotr Rudnicki. An Overview of SMURPH: an Object-oriented Configurable Simulator for Low-level Communication Protocols. Technical report, University of Alberta, Department of Computer Science, Edmonton, Alberta, October 1992.

[GR92b] Paweł Gburzyński and Piotr Rudnicki. The SMURPH Protocol Modeling Environment, version 1.61. Technical report, University of Alberta, Department of Computer Science, Edmonton, Alberta, September 1992.

[GR92c] Jens Grabowski and Ekkart Rudolph. Message Sequence Chart (MSC) - A Survey of the new CCITT Language for the Description of Traces within Communication Systems. University of Berne, Institute for Informatics and Applied Mathematics, IAM 92-022, 1992.

[Hai82] Brent T. Hailpern. *Verifying concurrent processes using temporal logic.* Springer-Verlag(Lecture notes in computer science. 129.), Berlin ; New York, 1982.

[Hea93] N. W. Heap. *An introduction to OSI.* Osney Meade, Oxford : Blackwell Scientific, Oxford, 1993.

[HG] Issam Hamid and Reinhard Gotzhein. Temporal Logic and its Applications. lecture series from November 6, 1992 to December 12, 1992.

[HGP92] G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Protocol specification, testing, and verification, XII*, pages 349–363, Lake Buena Vista, FL, USA, 1992. IFIP, Elsevier Science B.V. (North-Holland).

[Hola] Gerard J. Holzmann. Basic Spin Manual. A user's manual for SPIN users which comes with SPIN.

[Holb] Gerard J. Holzmann. Using SPIN—A Validator's Roadmap. A validator's manual for SPIN users which comes with SPIN.

[Hol85] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(10), December 1985. reprinted in: Current advances in distributed computation, Computer Science Press, 1986.

[Hol87] Gerard J. Holzmann. On Limits and Possibilities of Automated Protocol Analysis. In Harry Rudin and Colin H. West, editors, *Protocol specification, testing, and verification, VII*, pages 339–344. IFIP, Elsevier Science B.V. (North-Holland), 1987.

[Hol88] G.J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice & Experience*, 18(2):137–161, February 1988.

[Hol90a] Gerard J. Holzmann. Algorithms for Automated Protocol Verification. *AT&T Technical Journal*, 69(1):32–44, January/feb 1990. (Special Issue on Protcol Testing and Validation).

[Hol90b] G.J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2), February 1990. Special Issue on Protocol Testing, Specification, and Verification.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[Hol93] G.J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, April 1993.

[Hol95a] Gerard J. Holzmann. SPIN NEWS. An electronic news letter for SPIN users, February 1995.

[Hol95b] Gerard J. Holzmann. What's New in SPIN Version 2.0 (Draft). An updated addendum to the user's manual for SPIN users which comes with SPIN, January 1995.

[HP89] Gerard J. Holzmann and J Patti. Validating SDL specifications: an experiment. In Ed. Brinksma, Giuseppe Scollo, and Chris A. Vissers, editors, *Protocol specification, testing, and verification, IX*, pages 317–326, Enschede, The Netherlands, 1989. IFIP, Elsevier Science B.V. (North-Holland).

[ISOa] Information Processing Systems — Open System Interconnection. *ISO Draft International Standard 10167: Guidelines for the Application of Estelle, LOTOS and SDL.*

[ISOb] Information Processing Systems — Open System Interconnection. *ISO International Standard 9074: Estelle — A Formal Description Technique Based on an Extended State Transition Model.*

[ISOc] Information Processing Systems — Open System Interconnection. *ISO International Standard 8807: LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.*

[Lam82] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems*, 5(2):190–222, April 1982.

[Lam85] Leslie Lamport. Chapter 2: Basic Concepts. In M. Paul and H.J. Siegert, editors, *Lecture notes in computer science. 190.*, pages 6–43. the Institut fur Informatik, Technische Universitat Munchen, Springer-Verlag, 1985.

[Lin83] Peter F. Linington. Fundementals of the Layer Service Definitions and Protocol Specifications. *Proceedings of the IEEE*, 71(12):1341–1345, December 1983.

[Lin85] Huai-An Lin. *A new methodology for designing communication protocols*. PhD thesis, University of Ohio, 1985.

[Liu89] Ming T. Liu. Protocol Engineering. In *Advances in computers*, volume 29, pages 79–195, New York, 1989. Academic Press.

[LL92a] Peter B. Ladkin and Stefan Leue. An Analysis of Message Sequence Charts. University of Berne, Institute for Informatics and Applied Mathematics, IAM 92-013, June 1992.

[LL92b] Peter B. Ladkin and Stefan Leue. An Automaton Interpretation of Message Sequence Charts. University of Berne, Institute for Informatics and Applied Mathematics, IAM 92-012, June 1992.

[LPU90] Luigi Logrippo, Robert L. Probert, and Hasan Ural, editors. *Protocol specification, testing, and verification, X*, Ottawa, Ontario, Canada, 1990. IFIP, Elsevier Science B.V. (North-Holland).

[Lyn68] W. C. Lynch. Reliable Full-Duplex Transmission over Half-Duplex Telephone Lines. *Comm. of the ACM*, 11(6):406–410, June 1968.

[MGK91] Susan Murphy, Per Gunningberg, and John P.J. Kelly. Experiences with estelle, lotos and sdl: A protocol implementation. *Computer Networks and ISDN Systems*, 22:51–59, 1991.

[Mil90] Raymond E. Miller. Protocol Verification: The first ten years, the next ten years; some personal obserations. In Logrippo et al. [LPU90], pages 1–39.

[MP90] Zohar Manna and Amir Pnueli. Tools and Rules for the Practicing Verifier. Report STAN-CS-90-1321, Stanford University, July 1990.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, New York, 1992.

[MW84] Z. Manna and P. Wolper. Synthesis of communication processes from temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 6(1):68–93, January 1984.

[NA89a] D. H. New and P. D. Amer. Adding Graphics and Animation to Estelle. Technical Report 89–14, University of Delaware, January 1989.

[NA89b] D. H. New and P. D. Amer. Adding Graphics and Animation to Estelle. In E. Brinksma, C. Vissers, and G. Scollo, editors, *Participants proceedings of 9th IFIP WG 6.1, International Symposium on Protocol Specification, Testing, and Verification*, Enschede, The Netherlands, June 1989.

[NA91] D. H. New and P. D. Amer. *Protocol Visualization of Estelle Specifications.* North Holland, Amsterdam (in press), 1991.

[New91] D. H. New. *Protocol Visualization.* PhD thesis, University of Delaware, 1991.

[Ous94] John Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, Inc., 1994.

[Pet81] James Lyle Peterson. *Petri net theory and the modeling of systems.* Prentice-Hall, Englewood Cliff, N.J., 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. pages 46–57, October 1977. Proc. 18th annual Symposium on the Foundations of computer science.

[Rei85] Wolfgang Reisig. *Petri nets : an introduction.* Springer-Verlag, Berlin ; New York, 1985.

[Rei92] Wolfgang Reisig. *A primer in Petri net design.* Springer-Verlag, Berlin ; New York, 1992.

[SDW92] W.T. Strayer, B.J. Dempsey, and A.C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley Publishing Company, Inc., 1992.

[Sil92] Silicon Graphics Inc. and Protocol Engines Inc. *XTP Protocol Definition*, January 11 1992.

[SL93] Bernard Stepien and Luigi Logrippo. Status-Oriented Telephone Service Specification: An Exercise in LOTOS Style. University of Ottawa, Department of Computer Science, TR-93-07, February 1993.

[ST87] R. Saracco and P.A.J. Tilanus. Ccitt sdl: Overview of the language and its applications. *Computer Networks and ISDN Systems*, 13:65–74, 1987.

[Ste] D.E. Stevenson. 1001 Reasons for not Proving Programs Correct: A Survey. Technical report, Clemson University. Preprint for *Philosophy and Computers*.

[Tar91] Katie Tarnay. *Protocol specification and testing*. Plenum, New York ; London, 1991.

[Tur93] Kenneth J. (ed) Turner. *Using Formal Description Techniques - An introduction to Estelle, LOTOS and SDL*. John Wiley and Sons Ltd., West Sussex, England, 1993.

[Vis90] Chris A. Vissers. FDTs for Open Distributed Systems, a retrospective and a prospective view. In Logrippo et al. [LPU90], pages 1–22.

[Wel95] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, 1995.

[Wes89] C.H. West. Protocol validation in complex systems. *Computer Communication Review*, 19(4):303–12, September 1989.

[XTP95] XTP Forum, 1900 State Street, Santa Barbara CA 93101. *XTP Protocol Definition, Version 4.0*, March 1995.

Appendix A

Specification and Validation Approach

[Hol91] gives ten basic rules of protocol design, which we find useful. They are:

1. Make sure that the problem is well-defined. All design criteria, requirements and constraints, should be enumerated before a design is started.

2. Define the service to be performed at every level of abstraction before deciding which structures should be used to realize these services (*what* comes before *how*).

3. Design *external* functionality before *internal* functionality. First consider the solution as a black box and decide how it should interact with its environment. Then decide how the black-box can be internally organised. Likely it consists of smaller black-boxes that can be refined in a similar fashion.

4. Keep it simple. Fancy protocols are buggier than simple ones; they are harder to implement, harder to verify, and often less efficient. There are truly complex problems in protocol design. Problems that appear complex are often just simple problems huddled together. Our job as designers is to identify the simpler problem, separate them, and then solve them individually.

5. Do not connect what is independent. Separate orthogonal concerns.

6. Do not introduce what is immaterial. Do not restrict what is irrelevant. A good design is "open-ended," i.e., easily extendible. A good design solves a class of problems rather than a single instance.

7. Before implementing a design, build a high-level prototype ... and verify that the design criteria are met ....

8. Implement the design, measure its performance, and if necessary, optimise it.

9. Check that the final optimised implementation is equivalent to the high-level design that was verified.

10. Don't skip Rules 1 to 7.

and then he adds: The most frequently violated rule, clearly, is Rule 10.

Appendix B

TSD Source Code

The following Tcl/Tk programs constitute the TSD and XTSD programs used to produce TSD diagrams.

B.1 Main Program (text version)

file name: /home/ted/bin/tsd

```
 1  #!/usr/bin/tclsh
 2  #!/local/paths/tclsh
 3  if {$argc > 0} {
 4  set LANGUAGE  [lindex $argv 0]
 5  } else {
 6  set LANGUAGE  p2
 7  }
 8  set X 0
 9  if [file exists "./tsd.h"] { source "./tsd.h"
10          } else {puts stdout "warning: LOSS_LINE not set";set LOSS_LINE -1}
11  source "~/bin/tsd-lib.tcl"
12  #-------------------------------------------------------------------------
13  setup
14  incomplete_sends
15  print_out
16  #-------------------------------------------------------------------------
```

B.2 Main Program (X windows version)

file name: /home/ted/bin/xtsd

```
#!/usr/bin/wish -f
#!/local/paths/wish -f
# Tk 4.1
if {$argc == 1} {
set LANGUAGE  [lindex $argv 0]
} else {
set LANGUAGE  p2
}
global env; # declaration necessary?
# set LOSS_LINE 160
if [file exists "./tsd.h"] { source "./tsd.h"
        } else { puts stdout "warning: LOSS_LINE not set";
                 set LOSS_LINE -1}
source "~/bin/tsd-lib.tcl"
# bogus test, wish is executing this script ...
if [info exists env(DISPLAY)] { set X 1 } else { set X 0 }
if $X {
set auto_path "/home/ted/bin $auto_path"
set text_y0 0

wm title . "tsd 2 ps"
wm iconname . "tsd 2 ps"
wm minsize . 4 2
# wm grid $c ?baseWidth baseHeight widthInc heightInc?


set n 0

set f [frame .f -borderw 2 -relief raised]
set c [canvas $f.c \
        -height "20 c" \
        -width "18 c"
        ]

proc pb-incr {} {
        global c
        global n
        $c postscript -file ./tsd-p-[pid]-$n.ps
        incr n
}
proc plb-incr {} {
        global c
        global n
        $c postscript -rotate true -file ./tsd-l-[pid]-$n.ps
        incr n
```

```
46 }
47
48 set m [frame .m -relief raised]
49 set qb [button $m.q -text "Quit" -command "destroy ."]
50 set sb [button $m.s -text "Scale" -command "scale_win"]
51 set pb [button $m.p -text "Print" -command "pb-incr"]
52 set plb [button $m.pl -text "Print landscape" -command "plb-incr"]
53
54
55 # Create vertical scroll bar
56 scrollbar .vs -command "vScrollProc"
57
58 # Create horizontal scroll bar
59 #       The .h and .h.corner frames are only so that it meshes
60 #       nicely with the vertical scrollbar.
61 frame .h
62 scrollbar .h.s -ori hori -command "$c xview"
63 frame .h.corner -width [lindex [.h.s config -width] 4]
64
65
66 pack .h.s -side right -fill x -expand 1
67 pack .h.corner -side left
68
69 # Pack scrollbars
70 pack .h -side bottom -fill x
71 pack .vs -side left -fill y
72
73 # Pack text/canvas
74 pack $qb $pb $plb $sb -fill x -side right
75 pack $m -fill both
76 pack $f $c -fill both -expand 1
77 }
78
79 #-------------------------------------------------------------------------
80 setup
81 incomplete_sends
82 print_out
83 #-------------------------------------------------------------------------
84
85 if $X {
86 # Set the viewable area.  Must change this with the configuration of the
87 # canvas changes (i.e., it is resized)
88 $c config -width [winfo reqw $c] -height [winfo reqh $c]
89
90
91 # Sets the size of the horizontal scrolling region to include the width
92 # of the text widget.  Must reissue this when you change the width or
93 # height of the text widget.
```

```
94  $c config -scrollregion [$c bbox all]
95  }
```

B.3 Library Program

file name: /home/ted/bin/tsd-lib.tcl

```
 1  proc setup {} {
 2  global numbLines numbSends numbRecs
 3  global proc_col col_proc
 4  global procs
 5  global queues
 6  global time
 7  global X
 8  global LANGUAGE
 9  global is_test_module
10  global QUEUE_CHOOSE
11  global X_COL_SPACING
12
13  set numbLines 0
14  set numbSends 0
15  set numbRecs 0
16  set numbCompletes 0
17  set is_test_module 0
18  set QUEUE_CHOOSE -1
19  set layout [pwd]/layout.tsd
20  set X_COL_SPACING 4
21  set X_COL_SPACING 3
22
23  if [catch {open $layout r} layoutid] {
24   puts stderr "file [file tail $layout] does not exit: \
25                  creating [file tail $layout]"
26   set layoutid [open $layout w]
27   } else {
28   if {[file size $layout] > 0} {
29    while {[gets $layoutid line] >= 0} {
30     set loop 0
31     set numbElements [llength $line]
32  puts $numbElements
33     while {$loop < $numbElements} {
34       set col_proc($loop) [lindex $line $loop]
35       if ![info exists proc_col($col_proc($loop))] {
36         set proc_col($col_proc($loop)) $loop
37       } else {
38         incr loop
39         set QUEUE_CHOOSE [lindex $line $loop]
40         set is_test_module 1
41       }
42       incr loop
43     }; #while
44    }; #while
45   } else {
```

```
 close $layoutid
 set layoutid [open $layout w]
 }; #if
}; #if

 while {[gets stdin in_line] >= 0} {
   if {[string compare $LANGUAGE g] ==0} {
        set_arr_gen $in_line
   } elseif { [string compare $LANGUAGE p1] ==0} {
        set_arr_pre_2 $in_line
   } elseif { [string compare $LANGUAGE p2] ==0} {
        set_arr $in_line
   } else { puts "What language am I reading?"; exit }
#---------------------------------------

        if {[string compare $arr(action) Send] == 0} {
                lappend queues($arr(queue)) \
                        [list $arr(action) \
                        $arr(message) \
                        $arr(proc) \
                        $arr(queue) \
                        $arr(line) \
                        $numbLines]
                incr numbLines
                incr numbSends
        }
#---------------------------------------

        if {[string compare $arr(action) Recv] == 0} {
         set from_arr(action) [lindex [split \
                [lindex $queues($arr(queue)) 0]] 0]
         set from_arr(message) [lindex [split \
                [lindex $queues($arr(queue)) 0]] 1]
         set from_arr(proc) [lindex [split \
                [lindex $queues($arr(queue)) 0]] 2]
         set from_arr(queue) [lindex [split \
                [lindex $queues($arr(queue)) 0]] 3]
         set from_arr(line) [lindex [split \
                [lindex $queues($arr(queue)) 0]] 4]
         set from_numbLines [lindex [split \
                [lindex $queues($arr(queue)) 0]] 5]
         set queues($arr(queue)) [lreplace $queues($arr(queue)) 0 0]

if {$X} { x_draw_arrows } else {
         set time($from_numbLines) \
                [list \
                $from_arr(proc) \
                $from_arr(message) \
```

```
94                  $from_arr(queue) \
95                  ! \
96                  $numbCompletes
97                  ]
98          set time($numbLines) \
99                  [list \
100                 $arr(proc) \
101                 $arr(message) \
102                 $arr(queue) \
103                 ? \
104                 $numbCompletes
105                 ]
106 }; #if
107
108
109          incr numbLines
110          incr numbRecs
111          incr numbCompletes
112         }
113  }; #while
114  if {[file size $layout] == 0} {
115   set index 0
116   set numbCol_proc [array size col_proc]
117   while {$index < $numbCol_proc} {
118    puts -nonewline $layoutid $col_proc($index)
119    puts -nonewline $layoutid " "
120    incr index
121   }; #while
122   puts $layoutid " "
123  }
124  close $layoutid
125 }
126 #-----------------------------
127 proc set_arr {in_line} {
128 upvar arr arr layout layout proc_col proc_col col_proc col_proc
129 global procs
130         set arr(state) [lindex $in_line 0]
131         set arr([lindex $in_line 1]) [lindex $in_line 2]; #proc
132         set arr([lindex $in_line 4]) [lindex $in_line 5]; #line
133         set arr([lindex $in_line 10]) [lindex $in_line 11]; #queue
134         set arr(queue_name) [lindex $in_line 12]
135         set arr(file) [lindex $in_line 6]
136         set arr(action) [lindex $in_line 7]
137         set arr(message) [lindex $in_line 8]
138         set loop 0
139         if {[string compare $arr(action) Send] == 0 || \
140             [string compare $arr(action) Recv] == 0} {
141                 if ![info exists procs($arr(proc))] {
```

```
142                     set procs($arr(proc)) [lindex $in_line 3]
143                     set index [expr [array size procs] - 1]
144                     if {[file size $layout] == 0} {
145                          set col_proc($index) [lindex $in_line 2]
146                          if ![info exists proc_col($col_proc($index))] {
147                                  set proc_col($col_proc($index)) $index
148                          }
149                     }
150                 }
151         }
152 }
153 #-----------------------------
154 proc x_draw_arrows {} {
155 global QUEUE_CHOOSE LOSS_LINE c X_COL_SPACING
156 upvar proc_col proc_col arr arr from_arr from_arr
157 upvar from_numbLines from_numbLines numbLines numbLines
158 set numbProcs [array size proc_col]
159 set Y_OFFSET 1
160                 set x0 $proc_col($from_arr(proc))
161                 set x0 [expr $x0 * $X_COL_SPACING]
162                 set y0 $from_numbLines
163                 incr y0 $Y_OFFSET
164
165                 if { ([string compare $arr(queue) $QUEUE_CHOOSE] == 0)} {
166                         set x1 [expr $proc_col($arr(proc))+$numbProcs]
167                 } else {
168                         set x1 $proc_col($arr(proc))
169                 }
170                 set x1 [expr $x1 * $X_COL_SPACING]
171                 set y1 $numbLines
172                 incr y1 $Y_OFFSET
173 if {$x0<$x1} {
174 set xt [expr $x0 + ($x1 - $x0)/2.0]
175 } else {
176 set xt  [expr $x1 + ($x0 - $x1)/2.0]
177 }
178 set yt [expr ($y0 + ($y1 - $y0)/2.0)]
179 # puts $xt
180 # puts $yt
181 # set xt $x0
182 # set yt $y0
183 # set color for lines
184 #set depth [exec xwininfo -root | grep Depth | awk '{print $NF}']
185 set depth [lindex [exec xwininfo -root | grep Depth ] 1]
186 #puts $depth
187 #puts [lindex [split $from_arr(message) ","] 0]
188 if {$depth > 1} {
189   if {[llength [split $from_arr(message) ","] ] <= 1} {
```

```
190             set colour grey
191     } else {
192             set colour [lindex [split $from_arr(message) ","] 0]
193             if {[string compare $colour white] == 0} {
194                     set colour yellow
195             } elseif { [string compare $colour red] != 0 &&\
196                         [string compare $colour blue] != 0 } {
197                     set colour grey
198               }
199     }
200     $c create line "$x0 c" "$y0 c" "$x1 c" "$y1 c" -arrow last \
201     -fill  $colour
202   #puts $colour
203     $c create text "$xt c" "$yt c" \
204     -font -*-*-Bold-R-Normal--*-120-* \
205     -text $from_arr(message)
206   } else {
207     set colour black
208     $c create line "$x0 c" "$y0 c" "$x1 c" "$y1 c" -arrow last \
209     -stipple gray50 \
210     -fill  $colour
211   #puts $colour
212     $c create text "$xt c" "$yt c" \
213     -font -*-*-Bold-R-Normal--*-120-* \
214     -text $from_arr(message)
215   }
216   ### puts $arr(line)
217   if {[string trimright $arr(line) ,] == $LOSS_LINE} {
218    $c create bitmap "$x1 c" "$y1 c" -bitmap error
219   }
220   # update
221   }
222
223
224
225   #------------------------------
226   proc incomplete_sends {} {
227
228   global numbRecs numbSends time queues
229   global X
230
231                                     # puts $numbRecs
232                                     # puts $numbSends
233    if {$numbSends >  $numbRecs} {
234                                      puts [expr $numbSends - $numbRecs]
235   puts  [array names queues]
236           foreach arr_queue [array names queues] {
237                   while {[llength $queues($arr_queue)] > 0} {
```

```
238                     set from_arr(action) [lindex [split \
239                             [lindex $queues($arr_queue) 0]] 0]
240                     set from_arr(message) [lindex [split \
241                             [lindex $queues($arr_queue) 0]] 1]
242                     set from_arr(proc) [lindex [split \
243                             [lindex $queues($arr_queue) 0]] 2]
244                     set from_arr(queue) [lindex [split \
245                             [lindex $queues($arr_queue) 0]] 3]
246                     set from_arr(line) [lindex [split \
247                             [lindex $queues($arr_queue) 0]] 4]
248                     set from_numbLines [lindex [split \
249                             [lindex $queues($arr_queue) 0]] 5]
250                     set queues($arr_queue) [lreplace $queues($arr_queue) 0 0]
251
252   if {$X} { x_incomplete_sends } else {
253
254                     set time($from_numbLines) [list $from_arr(proc) \
255                            $from_arr(message) $from_arr(queue) ! \
256                            $from_numbLines dead]
257   }; #if
258                   }; #while
259           }; #foreach
260    }; #if
261
262   }
263
264
265   #-----------------------------
266   proc x_incomplete_sends {} {
267   global c proc_col col_proc X_COL_SPACING
268   upvar from_arr from_arr from_numbLines from_numbLines
269                     set x0 $proc_col($from_arr(proc))
270                     set x0 [expr $x0 * $X_COL_SPACING]
271                     set y0 $from_numbLines
272                     $c create bitmap "$x0 c" "$y0 c" -bitmap error
273                     $c create text "$x0 c" "$y0 c" \
274                            -font -*-Helvetica-Bold-R-Normal--*-140-* \
275                            -text $from_arr(message)
276   }
277   #-----------------------------
278
279   proc print_out {} {
280   global procs numbLines time proc_col col_proc QUEUE_CHOOSE is_test_module
281   global X
282   set numbProcs [array size procs]
283    set loop 0
284    set numbCols [expr $numbProcs + $is_test_module]
285    while {$loop < $numbCols} {
```

```
286          if {$X} {x_print_out $loop} else {
287                   if [info exists procs($col_proc($loop))] {
288                     puts -nonewline stdout "$procs($col_proc($loop))\t\t"
289                   }
290          }
291          incr loop
292   }; #while
293
294  if {!$X} {
295   puts stdout " "; #newline
296   set loop 0
297   while {$loop < $numbLines} {
298          # puts stdout "y:$loop = $time($loop)"
299          set yloop 0
300          set ytime($loop) ""
301          while {$yloop < $numbCols} {
302           append ytime($loop) "| "
303           incr yloop
304          }
305  # append ytime($loop) [list | | | | |]
306          # puts stdout $ytime($loop)
307
308   set col $proc_col([lindex $time($loop) 0])
309   set msg [lindex $time($loop) 1]
310   set que [lindex $time($loop) 2]
311   set act [lindex $time($loop) 3]
312   set last_v [expr [llength $time($loop)] - 1]
313   set val [lindex $time($loop) $last_v]
314   if {[string compare $val "dead"] == 0} { set asc_val $val
315   } else {
316   set asc_val [format "%c" [expr ([expr $val % 26])+65]]
317   };#[expr ([expr $val + 65])%([expr 65+26])]]
318          # set msg ""
319   set msg "$msg"
320   if {([string compare $act "?"] == 0) &&
321       ([string compare $que $QUEUE_CHOOSE] == 0)} {
322          set col [expr $col + $numbProcs]
323   }
324
325   set ytime($loop) [lreplace $ytime($loop) $col $col "$asc_val $msg"]
326
327  # puts stdout [lrange $ytime($loop) 0 [llength $ytime($loop)]]
328   puts stdout [join $ytime($loop) \t\t]
329  # puts stdout "y:$loop = $time($loop)"
330   incr loop
331   }; #while
332  }; #if
333  }
```

```
334
335
336 #-----------------------------
337 proc x_print_out {loop} {
338 global procs col_proc c numbLines X_COL_SPACING
339 ## draw the vertical lines and the column headers
340         set x0 [expr $loop * $X_COL_SPACING]
341         set y0 0.75; # 0.75
342         set x1 $x0
343         set y1 $numbLines
344         $c create text "$x0 c" "$y0 c" \
345                 -text "$procs($col_proc($loop))" \
346                 -anchor n \
347                 -font -*-Helvetica-Bold-R-Normal--*-120-* \
348                 -tags text_line
349         set y0 1
350
351         $c create line "$x0 c" "$y0 c" "$x1 c" "$y1 c" -width 2
352 }
353
354 #-----------------------------
355 proc set_arr_pre_2 {in_line} {
356 upvar arr arr layout layout proc_col proc_col col_proc col_proc
357 global procs
358
359         set arr(state) nil
360         set arr([lindex $in_line 0]) [lindex $in_line 1]; #proc
361         set arr([lindex $in_line 3]) [lindex $in_line 4]; #line
362         set arr([lindex $in_line 8]) [lindex $in_line 9]; #queue
363         set arr(queue_name) [lindex $in_line 10]
364         set arr(file) nil
365         set arr(action) [lindex $in_line 5]
366         set arr(message) [lindex $in_line 6]
367
368 #       puts stdout "$arr(action) $arr(message) $arr(proc) \
369 #                       $arr(queue) $arr(line)"
370
371         set loop 0
372         if {[string compare $arr(action) Send] == 0 || \
373             [string compare $arr(action) Recv] == 0} {
374                 if ![info exists procs($arr(proc))] {
375                   set procs($arr(proc)) [lindex $in_line 2]
376                   set index [expr [array size procs] - 1]
377                   if {[file size $layout] == 0} {
378                     set col_proc($index) [lindex $in_line 1]
379                     if ![info exists proc_col($col_proc($index))] {
380                        set proc_col($col_proc($index)) $index
381                     }
```

```
382                     }
383                 }
384         }
385
386 }
387
388 #-----------------------------
389 proc set_arr_gen {in_line} {
390 upvar arr arr layout layout proc_col proc_col col_proc col_proc
391 global procs
392
393         set arr(state) nil
394         set arr(proc) [lindex $in_line 0]
395         set arr(line) [lindex $in_line 4]
396         set arr(queue) [lindex $in_line 1]
397         set arr(queue_name) [lindex $in_line 1]
398         set arr(file) nil
399         set arr(action) [lindex $in_line 2]
400         if {[string compare $arr(action) "!"] == 0 } {
401                 set arr(action) Send
402         }
403         if {[string compare $arr(action) "?"] == 0 } {
404                 set arr(action) Recv
405         }
406         set arr(message) [lindex $in_line 3]
407
408 #       puts stdout "$arr(action) $arr(message) $arr(proc) \
409 #                       $arr(queue) $arr(line)"
410
411         set loop 0
412         if {[string compare $arr(action) Send] == 0 || \
413             [string compare $arr(action) Recv] == 0} {
414                 if ![info exists procs($arr(proc))] {
415                     set procs($arr(proc)) [lindex $in_line 0]
416                     set index [expr [array size procs] - 1]
417                     if {[file size $layout] == 0} {
418                        set col_proc($index) [lindex $in_line 0]
419                        if ![info exists proc_col($col_proc($index))] {
420                           set proc_col($col_proc($index)) $index
421                        }
422                     }
423                 }
424         }
425
426 }
427 #-----------------------------
428 proc scale_win {} {
429         global c
```

```
430         set scale_x .9
431         set scale_y .8
432         $c scale all 0 0 $scale_x $scale_y
433 # $c config -width [expr [winfo reqw $c]*$scale_x] \
434 #         -height [expr [winfo reqh $c]*$scale_y]
435         $c config  -scrollregion [$c bbox all]
436 }
437
438 #-----------------------------
439 proc vScrollProc index {
440         global c
441         global text_y0
442         global numbLines
443         $c yview $index
444         set win_height [expr ([winfo reqh $c]/[winfo fpixels $c "1c"])]
445         set win_height [expr round($win_height)]
446 # puts "---------"
447 # puts $numbLines
448 # puts $win_height
449         set last_window [expr ($numbLines - $win_height)]
450 # puts $last_window
451 # puts "index $index"
452 # puts "text_y0 $text_y0"
453         if {$index < 0 && $text_y0 <= 0}         {
454                 set text_y0 0
455         } elseif {$index < 0 && $text_y0 > 0}    {
456                 set text_y0 -$text_y0
457         } elseif {  ( $index > $last_window+1 && $index > $text_y0 )} {
458                 set text_y0 [expr $last_window - $text_y0]
459         } elseif {  ( $index > $last_window && $index > $text_y0 )} {
460                 set text_y0 0
461         } elseif {  ( $index > $last_window && $index <= $text_y0 )} {
462                 set text_y0 0
463         } else { set text_y0 [expr $index - ($text_y0)] }
464 # puts "move text_y0 $text_y0"
465         $c move text_line 0 ${text_y0}c
466         if {$index >= 0 && ($index <= $last_window)} {
467 # puts index
468         set text_y0 $index
469         } elseif { $index < 0} {
470         set text_y0 0
471         } else {
472 # puts last_window
473         set text_y0 [expr $last_window + 0]
474         }
475 # puts "new text_y0 $text_y0"
476 }
477
```

```
478  # need proc to force tag text_line to top of screen
```

B.4 Header Program

file name: /home/ted/bin/tsd.h

```
1  # not used - use tsd.h in current directory!
2  set LOSS_LINE 160
```

Appendix C

FSM Source Code

The following Tcl/Tk programs constitute the FSM and XFSM programs used to produce FSM diagrams.

C.1 Small Support Programs

```
file name: /home/ted/bin/fsm-ls
     1  pan -d | grep proctype | awk '{print $2}'
file name: /home/ted/bin/global-fsm
     1  pan -c0 |dag.awk|dag.tcl
file name: /home/ted/bin/pre-xdag
     1  #!/home/grad/tede/bin/dodrawsh
     2  dodrawing [lindex $argv 0]
file name: /home/ted/bin/line-only
     1  sed '1,$s/\.[0-9]*\ /\ /g'
file name: /home/ted/bin/empty-node
     1  awk '$1 != "du" {$1 = "{}"; print} $1 == "du" {print}'
     2  #awk '{ printf ("{} "); for (i=2;i<=NF;i++) printf ("%s ",$i);
     3  #printf ("\n");}'
```

file name: /home/ted/bin/dag.tcl

```
#!/local/paths/tclsh
proc set_arr {in_line} {
global arr
        set arr(type) [lindex $in_line 0]
        set arr([lindex $in_line 1]) [lindex $in_line 2]; #from
        set arr([lindex $in_line 3]) \
          [string trimright [lindex $in_line 4] \; ]; #to & remove ";"
}
#--------------------------------------
    # print elements of an array
    proc show_array arrayName {
        upvar $arrayName myArray

        foreach element [array names myArray] {
           set myArray($element) [join $myArray($element) \, ]
           puts stdout $element:$myArray($element)\;
        }
    }

#--------------------------------------
#    set arval(0) zero
#    set arval(1) one
#    show_array arval
#--------------------------------------
 while {[gets stdin in_line] >= 0} {
    if {[string match edge* $in_line] > 0} {
        set_arr $in_line

        if {[string compare $arr(type) edge] == 0} {
           if ![info exists tr($arr(from))] {
                set tr($arr(from)) $arr(to)
           } else {
                lappend tr($arr(from)) $arr(to)
           }
        }
    }
}
show_array tr
#--------------------------------------
```

file name: /home/ted/bin/exp.awk

```
     1  #!/bin/sh
     2  sed '1,$s/
//g' $* |
     3  awk '
     4  BEGIN { print "#!/local/paths/expect -f";
     5          print "spawn spin -i test-fc.pr" }
     6  /Select\ \[.*\]:*/ {
     7  printf ("expect {");
     8  for (i=1;i<NF;i++) printf ("%s ",$i);
     9  printf ("}\n");
    10  printf ("exp_send \"")
    11  printf ("%s\\n\"\n",$i)
    12  }
    13  END {print "interact"}
    14  ' > $*.inter
    15  chmod +x $*.inter
```

C.2 Main Program (layout calculation)

file name: /home/ted/bin/fsm-make

```
#!/local/paths/tclsh
# invoke program as 'fsm-make' for full state graph coordinates and as
# 'fsm-atomic' for compressed state machine
# (redraw graph so that all atomic transitions are telescoped
# into one transition)
set dist(0) 0
set line_no(0) 0
set actions(0) 0
set children(0) {}
set MaxDist 0

proc findfsm {pfind} {
 global children line_no actions MaxDist dist
 global TLABEL FORMAT argv0

 if {[string compare [file tail $argv0] fsm-atomic] == 0} {
  set AT 0
  set lastAT 0
  set lastI 0
  set atoms(0) {}
  set achildren(0) {}
  set aacnt(0) 0
  set oldsrc 0
 }

 set src 0; set trn 0; set trg 0
 set Want 0

 catch { foreach el [array names state] { unset state($el) } }
 catch { foreach el [array names children] { unset children($el) } }
 catch { foreach el [array names dist]  { unset dist($el) } }

 if {[file exists pan]} {
  set fd [open "|./pan -d" w+]
 } else {
  puts "$argv0 aborted --- compile analyser with SPIN first!"
  return
 }
 set MaxDist 0
 while {[gets $fd line] > -1} {
  set key [lindex $line 0]
  if {[string compare $key proctype] == 0} {
   if { $Want == 1 } {
     break
   }
```

```
46      if {[string compare [lindex $line 1] $pfind] == 0} {
47       set Want 1
48       set dist($src) 0
49       set line_no($src) "0"
50       set achildren($src) {}
51       set aactions($src) "-nada-"
52
53  #    puts -nonewline $key
54  #    puts [lindex $line 1]
55  #    puts "dist($src): $dist($src)"
56  #    puts "line_no($src): $line_no($src)"
57  #    puts "achildren($src): $achildren($src)"
58  #    puts "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
59  #    puts $Want
60  #    puts [lindex $line 1]
61      }
62     } elseif { $Want == 1 \
63     && [string compare $key state] == 0} {
64      scan $line " state %d -(tr %d)-> state %d" \
65      src trn trg
66  #   puts "$src $trg"
67      set k [string first "line" $line]
68      if { $k > 0 } {
69       set m [string first "=>" $line]
70       incr k 5
71       incr m -2
72       set lbl [string range $line $k $m]
73  #    puts $lbl
74       incr m 4
75       set action [string range $line $m end]
76      }
77      set line_no($src) $lbl
78  #   puts "line_no($src): $line_no($src)"
79      if { [info exists dist($src)] == 0 } {
80       set dist($src) 0
81       set achildren($src) {}
82      }
83      if { [info exists dist($trg)] == 0 } {
84       set dist($trg) [expr $dist($src) + 1]
85       set achildren($trg) {}
86       if {$dist($trg) > $MaxDist } {
87        set MaxDist $dist($trg)
88       }
89      } else {
90       if { [expr $dist($src) + 1] < $dist($trg) } {
91        set dist($trg) [expr $dist($src) + 1]
92        if {$dist($trg) > $MaxDist } {
93         set MaxDist $dist($trg)
```

```
94        }
95       }
96      }
97      lappend achildren($src) $trg
98      lappend aactions($src) $action
99      lappend aparents($trg) $src
100 #   puts "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
101 #   puts "src: $src"
102 #   puts "children: $achildren($src)"
103 #   puts "actions: $aactions($src)"
104 #   puts "trg's parents: $aparents($trg)"
105    };# "state" (elseif)
106   };# while
107
108
109 #puts "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
110   if {[string compare [file tail $argv0] fsm-atomic] == 0} {
111    set Want 0
112    set fd [open "|./pan -d" w+]
113    while {[gets $fd line] > -1} {
114     set key [lindex $line 0]
115     if {[string compare $key proctype] == 0} {
116      if { $Want == 1 } {
117       break
118      }
119      if {[string compare [lindex $line 1] $pfind] == 0} {
120       set Want 1
121       set bumpy 1
122      }
123     } elseif { $Want == 1 \
124     && [string compare $key state] == 0} {
125      scan $line "  state %d -(tr %d)-> state %d" \
126      src trn trg
127
128      set j [string first "\]" $line]
129      set j0 [expr $j+3]
130      set j1 [expr $j+3]
131      if \
132      {[ string compare "A" \
133       [string range $line $j0 $j1]
134       ] == 0 } { set AT 1 } else { set AT 0
135      }
136      lappend atoms($src) $AT
137 #    puts [string range $line $j0 $j1]
138 #     puts ($src)
139 #     puts $atoms($src)
140 #    puts ATOMIC
141
```

```
142  #   find atomicity of parent
143      if {[info exists achildren] != 0} {
144       foreach srcpar [array names achildren] {
145  #     puts $srcpar
146        set ind [lsearch -exact $achildren($srcpar) $src]
147  #     puts $ind
148  #      if {$ind > -1} { puts "pchildren: $achildren($srcpar)"}
149        if {$ind > -1 && [info exists atoms($srcpar)] != 0} {
150         set lastI [lindex $atoms($srcpar) $ind];
151  #      puts $srcpar;
152         set par $srcpar;
153  #      puts found;
154         break
155        }
156       }; #foreach
157      }
158      if !$AT {
159       if {$lastI == 0 || $ind < 0} {
160  ###      puts NN
161  #     puts "src>dist($src): $dist($src)"
162  #     puts "MaxDist: $MaxDist"
163        set children($src)  $achildren($src)
164  #     puts "children($src): $children($src)"
165        set actions($src)  $aactions($src)
166  #     puts "actions($src): $actions($src)"
167  #     puts "trg>dist($trg): $dist($trg)"
168  #     puts "------------------------"
169       } else {
170  ###      puts AN
171        set atrg $trg
172        set children($src)  $achildren($src)
173  #     puts "children($src): $children($src)"
174        set actions($src)  $aactions($src)
175       }
176       #!$ATOMIC
177       set bumpy 1
178      } else {
179       if {$lastI == 0 || $ind < 0} {
180  ###      puts NA
181        set children($src)  $achildren($src)
182        set actions($src)  $aactions($src)
183        set asrc $src
184        set atrg $trg
185        set line_no($asrc) $line_no($src)
186        set children($asrc)  $achildren($src)
187        set actions($asrc)  $aactions($src)
188        set aacnt($trg) 0
189        set bumpy 1
```

```
190       } else {
191   ###     puts AA
192         if {[llength $achildren($src)] > 1 || \
193             [llength $aparents($trg)] > 1} {
194          set children($src)  $achildren($src)
195          set actions($src)  $aactions($src)
196          set bumpy 1
197         } else {
198          if $bumpy {
199   ###        puts bumpy
200           if {[llength $aparents($src)] == 1} {
201            if {[llength $achildren($aparents($src))] > 1} {
202             set indind [lsearch -exact $achildren($aparents($src)) $src]
203             set asrc $aparents($src)
204            } else {
205             set indind [lsearch -exact $src $src]
206             set asrc $src
207            }
208           } else {
209            set indind [lsearch -exact $src $src]
210            set asrc $src
211           }
212          # bumpy
213          }
214          if {[info exists children($asrc)] == 0} {
215           set children($asrc) $trg
216           set actions($asrc) $aactions($trg)
217          } else {
218   #ted if [info exists indind]  puts $indind
219           set children($asrc) \
220            [lreplace $children($asrc) $indind $indind $trg]
221           set actions($asrc) \
222            [lreplace $actions($asrc) $indind $indind $aactions($src)]
223          }
224          set bumpy 0
225   ###       puts -nonewline "children of ancestor: "
226   ###       puts ($asrc)
227   ###       puts $children($asrc)
228   ###       puts $actions($asrc)
229   ###       puts "---------------"
230         }
231          #$ATOMIC
232          set oldsrc $src
233        }
234       };# NN AN NA AA
235       #lappend parents($trg) $src)
236   ###    puts "$src -> $trg"
237   ###    if {[info exists children($src)] != 0} {puts $children($src)}
```

```
238 ###    if {[info exists line_no($src)] != 0} {puts $line_no($src)}
239 ###    if {[info exists actions($src)] != 0} {puts $actions($src)}
240 ###    puts "----------------------"
241     }
242    } ; #while
243   } else {
244    foreach i [array names achildren] {
245     set children($i) $achildren($i)
246     set actions($i) $aactions($i)
247    }
248    set Want 1
249   }
250
251
252
253   if {[string compare $FORMAT pre] == 0} {
254    foreach el [array names children] {
255     if {[llength $children($el)] != 0} {
256     if [info exists line_no($el)] { puts -nonewline $line_no($el)}
257      puts -nonewline "."
258      puts -nonewline $el
259
260      puts -nonewline ":"
261      for {set i 0} {$i < [llength $children($el)]} {incr i} {
262       puts -nonewline "$line_no([lindex $children($el) $i])"
263       puts -nonewline "."
264       puts -nonewline "[lindex $children($el) $i]"
265       if {[string compare $TLABEL tl] == 0} {
266        puts -nonewline " {[lindex $actions($el) $i]}"
267       }
268       if {$i < [expr [llength $children($el)] -1]} {
269        puts -nonewline ","
270       }
271      }
272      puts ";"
273     };# foreach
274    }
275   } elseif { [string compare $FORMAT x] == 0} {
276    if { $Want == 1 } {
277     dograph $pfind
278    } else {
279     puts "sorry, $pfind not found..."
280    }
281   }
282   catch "close $fd"
283  }
284  #---------------------------------
285  proc dograph {n} {
```

```
286         global dist children line_no actions MaxDist
287         global TLABEL FORMAT
288         set count -1
289
290
291         foreach el [array names dist]  {        # for every state
292           for {set i 0} { [lindex $children($el) $i] != "" } {incr i} {
293                 set trg [lindex $children($el) $i]
294                 if { $dist($trg) < $dist($el) } {
295                   set NotMiddle($el) 1
296 #puts "NotMiddle($el)"
297         } }     }
298
299 set y -85
300 set x_start  400
301 set x_offset 100
302 set y_offset 100
303         while {$count < $MaxDist} {
304                 incr count
305                 incr y $y_offset
306                 set x $x_start
307                 foreach el [array names dist] {
308                         if {$dist($el) == $count} {
309                           if { [info exists NotMiddle($el)] != 0 } {
310                             if { $x == $x_start } {
311                               incr x $x_offset
312                             }
313                           }
314                           set state($el) [list $line_no($el) $el $x $y]
315 #puts $state($el)
316                           if { $x > $x_start } {
317                                       set x [expr $x - $x_start]
318                                       set x [expr $x_start - $x]
319                           } else {
320                                       set x [expr $x_start - $x]
321                                       incr x $x_offset
322                                       set x [expr $x_start + $x]
323                         }
324                       }
325                 }
326         }
327 foreach el [array names dist] {
328                 puts -nonewline "[lindex $state($el) 0]."
329                 puts -nonewline "[lindex $state($el) 1] "
330                 puts -nonewline "[lrange $state($el) 2 end]"
331                   set tmp {}
332                   for {set i 0} { [lindex $children($el) $i] != "" }
333                       {incr i} {
```

```
334                         set trg [lindex $children($el) $i]
335                         set lbl  [lindex $actions($el) $i]
336                         set fromx [lindex $state($el) 2]
337                         set fromy [lindex $state($el) 3]
338                         set tox [lindex $state($trg) 2]
339                         set toy [lindex $state($trg) 3]
340 #puts -nonewline "< [lindex $actions($el) $i] >"
341                         if {[string compare $TLABEL tl] == 0} {
342                          set tlbl [lindex $actions($el) $i]
343                         }
344                         incr fromx 15
345                         incr tox 15
346                         if {$fromy < $toy} {
347                          incr fromy 30
348                         } else {
349                          incr toy 30
350                         }
351                         lappend tmp "$fromx $fromy $tox $toy"
352                         if {[string compare $TLABEL tl] == 0} {
353                          lappend tmp "{$tlbl}"
354                         }
355                  }
356                 puts " $i [join $tmp]"
357 #puts "---------------------------------------"
358 }
359 }
360 #--------------------------------
361
362 if {$argc==3} {
363   set TLABEL [lindex $argv 2]
364   set FORMAT [lindex $argv 1]
365 } elseif  {$argc==2} {
366     set TLABEL tu
367     set FORMAT [lindex $argv 1]
368   } else {
369     set TLABEL tu
370     set FORMAT x
371   }
372 #puts $TLABEL
373 #puts $FORMAT
374 findfsm [lindex $argv 0]
```

C.3 Main Program (draw graphic (X windows))

file name: /home/ted/bin/xdag

```
#!/usr/bin/wish -f
###!/local/paths/wish -f
# Tk 4.1
source "~/bin/dag-lib.tcl"
# bogus test, wish is executing this script ...
if [info exists env(DISPLAY)] { set X 1 } else { set X 0 }
if $X {
set auto_path "/home/grad/tede/bin $auto_path"

wm title . "dag 2 ps"
wm iconname . "dag 2 ps"
wm minsize . 4 2
# wm grid $c ?baseWidth baseHeight widthInc heightInc?


set n 0

set f [frame .f -borderw 2 -relief raised]
set c [canvas $f.c \
        -height "20 c" \
        -width "18 c"
        ]

proc pb-incr {} {
        global c
        global n
        $c postscript -file ./dag-p-[pid]-$n.ps
        incr n
}
proc plb-incr {} {
        global c
        global n
        $c postscript -rotate true -file ./dag-l-[pid]-$n.ps
        incr n
}

set m [frame .m -relief raised]
set qb [button $m.q -text "Quit" -command "destroy ."]
set sb [button $m.s -text "Scale" -command "scale_win"]
set pb [button $m.p -text "Print" -command "pb-incr"]
set plb [button $m.pl -text "Print landscape" -command "plb-incr"]


# Create vertical scroll bar
scrollbar .vs -command "$c yview"
```

```
46
47  # Create horizontal scroll bar
48  #       The .h and .h.corner frames are only so that it meshes
49  #       nicely with the vertical scrollbar.
50  frame .h
51  scrollbar .h.s -ori hori -command "$c xview"
52  frame .h.corner -width [lindex [.h.s config -width] 4]
53
54
55  pack .h.s -side right -fill x -expand 1
56  pack .h.corner -side left
57
58  # Pack scrollbars
59  pack .h -side bottom -fill x
60  pack .vs -side left -fill y
61
62  # Pack text/canvas
63  pack $qb $pb $plb $sb -fill x -side right
64  pack $m -fill both
65  pack $f $c -fill both -expand 1
66  }
67
68  #-------------------------------------------------------------------------
69  setup
70  #incomplete_sends
71  #print_out
72  #-------------------------------------------------------------------------
73
74  if $X {
75  # Set the viewable area.  Must change this with the configuration of the
76  # canvas changes (i.e., it is resized)
77  $c config -width [winfo reqw $c] -height [winfo reqh $c]
78
79
80  # Sets the size of the horizontal scrolling region to include the width
81  # of the text widget.  Must reissue this when you change the width or
82  # height of the text widget.
83  $c config -scrollregion [$c bbox all]
84  }
```

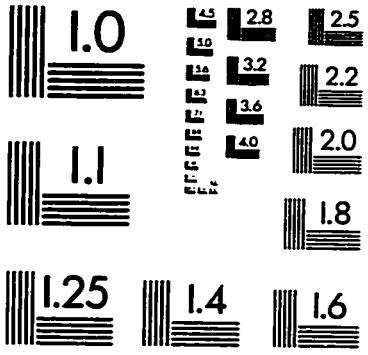
C.4 Library Program

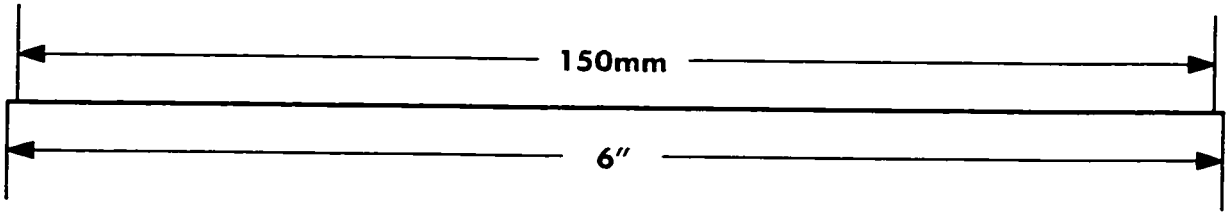
file name: /home/ted/bin/dag-lib.tcl

```
proc setup {} {
 while {[gets stdin in_line] >= 0} {

        set_arr $in_line
 }; #while
}
#-----------------------------
proc set_arr {in_line} {
global c
upvar arr arr
        set arr(node) [lindex $in_line 0]
        set arr(nodex) [expr [lindex $in_line 1] ]
        set arr(nodey) [expr [lindex $in_line 2] ]
        set arr(num_edges) [lindex $in_line 3]
   if {[string compare $arr(node) du] !=0} {
          $c create oval "$arr(nodex)" "$arr(nodey)"  \
                "[expr $arr(nodex)+30]" "[expr $arr(nodey)+30]" \
          -fill  ""
          $c create text "[expr $arr(nodex)+15]" "[expr $arr(nodey)+15]" \
                -text $arr(node)
}
set loop 0
set base 4
        while {$loop< $arr(num_edges)} {
                set x0 \
                [expr [lindex $in_line [expr 4+($loop*$base) + 0] ] ]
                set y0 \
                [expr [lindex $in_line [expr 4+($loop*$base) + 1] ] ]
                set x1 \
                [expr [lindex $in_line [expr 4+($loop*$base) + 2] ] ]
                set y1 \
                [expr [lindex $in_line [expr 4+($loop*$base) + 3] ] ]
                $c create line "$x0" "$y0" "$x1" "$y1" -arrow last \
                -fill  black

if {[lindex $in_line [expr 4+($loop*$base) + 4] ] != ""} {
  if {[regexp {[^0-9+]+} [lindex $in_line [expr 4+($loop*$base) + 4] ]]} {
    set lt [lindex $in_line [expr 4+($loop*$base) + 4] ]
    if {$x0<$x1} {
     set xt [expr $x0 + ($x1 - $x0)/2.0]
    } else {
     set xt  [expr $x1 + ($x0 - $x1)/2.0]
    }
    set yt [expr ($y0 + ($y1 - $y0)/2.0)]
                $c create text "$xt" "$yt" \
```

```
46                  -font -*-Helvetica-Bold-R-Normal--*-100-* \
47                          -text $lt
48      set base 5
49      #puts $lt
50    } else {
51     set base 4
52    }
53  }
54            incr loop
55  # update
56            }
57  }
58  #------------------------------
59  proc scale_win {} {
60          global c
61          set scale_x .9
62          set scale_y .9
63          $c scale all 0 0 $scale_x $scale_y
64  # $c config -width [expr [winfo reqw $c]*$scale_x] \
65  #          -height [expr [winfo reqh $c]*$scale_y]
66          $c config  -scrollregion [$c bbox all]
67  }
68
69  #-----------------------------
```

TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989