



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-55324-3

Canada

The University of Alberta

VERIFYING COMMUNICATION PROTOCOLS IN MIZAR-2

by

John P. Adria

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1989

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: John P. Adria

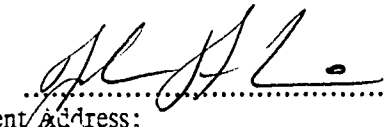
TITLE OF THESIS: Verifying Communication Protocols in MIZAR-2

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1989

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

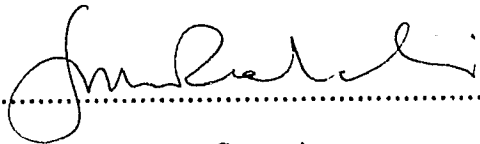
(Signed) 
Permanent Address:
10615-50 Street
Edmonton, Alberta
Canada T6A 2C9

Dated 10 October, 1989

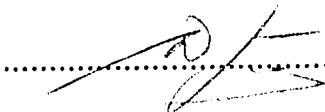
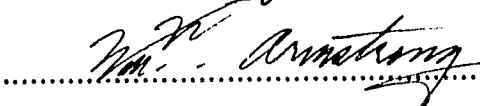
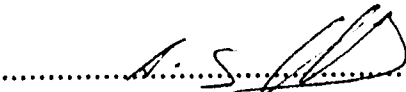
THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Verifying Communication Protocols in MIZAR-2** submitted by John P. Adria in partial fulfillment of the requirements for the degree of **Master of Science**.



Supervisor



Date: Oct. 10, 1989

ABSTRACT

This report describes a specification language and a proof technique for the formal specification and verification of computer communication protocols in MIZAR-2.

The specification of a protocol is recorded in MIZAR-2. Liveness is proved for perfect communication channels while safety is proved for imperfect channels, both using the MIZAR-2 proof recording language. We postulate and discuss conditions under which these two proofs together comprise a complete proof of safety and liveness for imperfect communication channels.

Acknowledgements

I wish to acknowledge the role of my supervisor, Professor Piotr Rudnicki, for the small bit of his immense knowledge of formal proofs that I have reproduced here. I also wish to acknowledge the operating system development group at IDACOM Electronics; everything useful I know about protocols, I learned there.

Most of all I wish to acknowledge and thank Kathryn, my wife, who has supported me in this and many other endeavours.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Overview	1
1.2. Communication Protocols	3
1.3. Properties to Be Verified	3
1.4. Formal Verification of Communication Protocols	4
1.5. The Specification Language	5
1.6. Method of Proof	6
Chapter 2: Survey of Specification and Verification Techniques	7
2.1. Overview	7
2.2. Significant Event Temporal Logic (SIGETL)	8
2.2.1. Overview	9
2.2.2. Structure of A SIGETL Specification	10
2.2.3. Remarks on the Semantics of SIGETL	10
2.2.4. Verification Using SIGETL	11
2.3. Modular Verification of Protocols	12
2.3.1. Overview	12
2.3.2. Structure of A Modular Protocol Specification	13
2.3.3. Semantics of A Modular Specification	13
2.3.4. Verification of a Modular Specification	14
2.4. Verification of Protocol Properties Using State Enumeration	14
2.4.1. Overview	14
2.4.2. Structure of Protocol Specification Using Blumer's Method	15
2.4.3. Semantics of a Specification Using Blumer's Method	16
2.4.4. Verification of Protocol Properties Using Blumer's Method	16
2.5. Reachability Analysis Using VALIRA	17
2.5.1. Overview	17
2.5.2. Structure of a VALIRA Specification	18
2.5.3. Analysis Performed by VALIRA	18
2.6. Protocol Specification Using Petri Nets	19
2.6.1. Overview	19
2.6.2. Structure of a Petri Net Specification	21
2.6.3. Semantics of a Petri Net Protocol Specification	21
2.6.4. Verification Using Petri Nets	22
Chapter 3: The Protocol Description Language	23
3.1. Introduction	23

3.2. Overview of PDL	23
3.3. PDL Syntax	23
3.3.1. Protocol Specification Structure	24
3.3.2. Channel Specification Structure	25
3.3.3. Party Specification Structure	26
3.3.4. State Specification Structure	28
3.3.5. Event Specification Structure	30
3.3.6. Statements	31
Chapter 4: PDL Semantics In MIZAR-2	33
4.1. Overview	33
4.2. Description of MIZAR-2	34
4.3. The Alternating Bit Protocol	37
4.4. PDL Semantics	38
4.4.1. Basic Objects	40
4.4.2. Time	41
4.4.3. Local Variables	41
4.4.4. Protocol Messages	42
4.4.5. Control Points	44
4.4.6. Channels	46
4.4.7. Events	48
4.5. Axiomatization of PDL Statements in MIZAR-2	49
4.5.1. Overview	49
4.5.2. Selection of Control Points	50
4.5.3. Sequencing Control	51
4.5.4. Generation of Axioms for Each Statement	51
4.5.4.1. Read_Client	51
4.5.4.2. Next_State	52
4.5.4.3. Transmit	53
4.5.4.4. Write_Client	55
4.5.5. MIZAR-2 Text For Wait_Event Statement	55
Chapter 5: Verification Procedure	59
5.1. Overview	59
5.2. On the Use of Temporal Logic	59
5.3. Required Safety and Liveness Proofs	60
5.3.1. Safety and Liveness Revisited	61
5.3.2. A Protocol's System State	61
5.3.3. Protocol Procedures	62
5.4. Modular Proof Structure	65
5.4.1. Lemmas in MIZAR-2	65

5.4.2. Lemma Structure	65
5.5. Verification of Safety	66
5.6. Verification of Liveness	66
5.7. Verification of Safety in the Alternating Bit Protocol	68
5.8. Verification of Liveness in the Alternating Bit Protocol	72
Chapter 6: Conclusion	77
6.1. Summary and Conclusions	77
6.2. Further Areas of Study	78
References	79
A1: Extensions To MIZAR-2	82
1.1. Lexical Modifications	82
1.2. Notational Modifications	83
1.2.1. Function Application Notation	83
1.2.2. Alternative Form of Predicate Definitions	83
A2: Example PDL Specification	86
A3: Alternating Bit Protocol in MIZAR-2	89
3.1. Introduction	89
3.2. Symbols Used In This Specification	89
3.2.1. Time	89
3.2.2. Local Variables	89
3.2.3. Protocol Messages	90
3.2.4. Control Points	90
3.2.5. Channels	91
3.3. Outline	91
A4: Safety Proof for Alternating Bit Protocol	100
4.1. Overview	100
4.2. Proof Structure	100
A5: Liveness Proof for Alternating Bit Protocol	111
5.1. Introduction	111
5.2. Proof Structure	111
A6: Syntax of PDL	121

List of Figures

Figure	Page
3.1 Protocol Definition Block	24
3.2 Channel Description Section Syntax	25
3.3 Channel Description from the Alternating Bit Protocol	25
3.4 Party Description	26
3.5 Party Init Section from the Alternating Bit Protocol	27
3.6 State Definition	28
3.7 State Initialization Specification	28
3.8 Structure of An Event Specification	30
3.9 Reception Event Specification	30
3.10 Example of the Use of Some PDL Statements	31
3.11 Possible Dialogue	31
4.1 PDL Concepts Recorded in MIZAR-2	40
4.2 MIZAR-2 Declaration of PDL Local Variables	42
4.3 MIZAR-2 Definitions for Messages	42
4.4 MIZAR-2 Definition of Unique Control Bits	43
4.5 ControlPoint Definition in MIZAR-2	44
4.6 PDL Transmit Statement Recorded in MIZAR-2	45
4.7 Excluded Middle Definition of Channels in MIZAR-2	46
4.8 PDL Communication Channels Recorded in MIZAR-2	46
4.9 Definition of Transmit Statement in MIZAR-2	47
4.10 Definition of Transmit Statement in MIZAR-2	48
4.11 ACK1 Reception and Message Transition in MIZAR-2	48
4.12 Three Pascal Statements And Their Axiomatizations	50
4.13 Axiomatization of <code>read_client</code>	51
4.14 Axiomatization of Transmit	53
4.15 <code>MBA_For_A_Unchanged</code> and <code>MAB_For_B_Unchanged</code>	54
4.16 Axiomatization of <code>write_client</code>	55
4.17 <code>Wait_Event</code> Example	56
4.18 Axiomatization of No Event Occurring	56
4.19 Axiomatization of Transition on Event Reception	57
4.20 Transition on Unspecified Event Reception	58
5.1 Relationship Between Three Temporal Operators	59
5.2 Representation of Temporal Operators Using Predicate Logic	60
5.3 Proposed Rule	60
5.4 Elements of the System State	62

5.5 System State Transition Graph	63
5.6 Steps of a Safety Proof	66
5.7 Safety Transition Lemma in MIZAR-2	69
5.8 Safety Transition Response of Party B in MIZAR-2	70
5.9 Party A Safety Response in MIZAR-2	71
5.10 Conclusion of Party B Safety Response in MIZAR-2	71
5.11 Definition of Predicate Prefix	72
5.12 General Structure of Liveness Proof	73
5.13 Liveness Transition Assumptions and Conclusion in MIZAR-2	74
5.14 Response of B in Liveness Procedure in MIZAR-2	75
5.15 Response of A in Liveness Procedure in MIZAR-2	75
A1.1 Predicate Ready_To_Go in Standard MIZAR-2	83
A1.2 Predicate Ready_To_Go Using Pre-Processor	85

Chapter 1

Introduction

1.1. Overview

This thesis describes a specification language and a proof technique for the formal specification and verification of communication protocols.

A protocol is specified in the specification language. Liveness is proved for perfect communication channels while safety is proved for imperfect channels. A proof of liveness demonstrates that work done by the protocol progresses. A proof of safety demonstrates that the protocol does not operate incorrectly in the face of communication errors. We argue that these two proofs together comprise a complete proof of safety and liveness for perfect and imperfect channels.

This thesis is divided into six chapters and six appendices.

The first chapter is this introduction and consists of an overview of the specification language and the proof technique.

The second chapter is a discussion of five other approaches for formal specification and verification of communication protocols. The five approaches were chosen because they use similar notions of the nature of protocols and verify correctness at the same level of detail.

The third chapter is a description of the specification language, called protocol description language(PDL). A superset of PDL is used in practice. The syntax of PDL is described and formally specified using the BNF notation. The semantics of PDL are

defined using the MIZAR-2 language.

The fourth chapter is a description of the MIZAR-2 language. MIZAR-2 is a formal language for recording mathematical statements and their proofs. There are a family of language processors available to check proofs recorded in MIZAR-2.

The fifth chapter is a description of the proof technique. The proof is constructed in MIZAR-2 using modular verification techniques. Small examples are given from the longer proofs contained in the appendices. We argue that proving liveness for perfect channels and proving safety for imperfect channels, constitutes a complete proof for safety and liveness for perfect and imperfect channels.

The sixth chapter contains the conclusions and directions for further research and study.

The first appendix describes extensions to MIZAR-2 to support verifications of PDL specifications.

The second appendix contains the example specification of the alternating bit protocol.

The third appendix is the full text of the description of the example specification in MIZAR-2

The fourth and fifth appendices are the full text of the proofs of safety and liveness for the example specification.

The sixth appendix is the full syntax of PDL.

1.2. Communication Protocols

A protocol provides communication between two or more different communicating parties, each of which may be implemented by different hardware. The communication protocol is the point at which the parties interface. The objects involved in interfacing the communicating parties must agree on the protocol (create a specification) and ensure that the agreement is kept by each party (validate the implementation).

Thus, the implementation of the protocol must be validated against a specification. The immense importance of correctness of protocols has resulted in a multi-billion dollar industry for the testing, validation and verification of protocols. The properties tested by the validation procedure depend on the application and nature of the protocol. This thesis is not about the validation of protocols against specifications. We discuss the verification of basic desirable properties of the protocol specifications themselves.

1.3. Properties to Be Verified

Our verification technique ensures that the basic properties of safety and liveness are present. These properties are accepted as being basic to any useful protocol and have been discussed extensively in the literature [BoS80, HaO83, Mer79, Sha86a, Sha86b, Sun79, TsV88].

Hailpern [HaO83] describes safety as "bad things will not happen" and points out that this is analogous to partial correctness proofs of sequential programs. Safety means that the protocol will not produce incorrect results including delivery of a message that was not sent, re-order the messages, or lose a message without knowledge that it was lost. Safety, like partial correctness, is not the only property which we would like to have proved.

Hailpern describes liveness as "good things will happen". Liveness means that at some point in the future, the protocol will correctly deliver a message or messages from one party to another. The exact definition of liveness depends on the service definition of the partic-

ular protocol. Another name for liveness is progress.

1.4. Formal Verification of Communication Protocols

In general, it is agreed that correct specifications are desirable. The validation of a protocol implementation against an incorrect specification is not necessarily a useful goal. Our motivation for ensuring the correctness of communication protocols is that a correct specification is the first step towards a correct implementation.

We choose to use manual formal verification (i.e. the use of human-created proofs to ensure correctness) as opposed to testing based techniques. There are two main reasons for the use of formal verification over testing.

The first reason is that testing can only determine the presence of errors, but not the absence of errors or actual correctness. Nor, in general, can testing ensure the presence of specific properties in all cases. Formal verification can determine that a specification is error free and that it does have desired properties.

The second reason is that we have observed that the manual formal verification process leads to an increased understanding of the nature of the specification [RuD85,Rud87].

As discussed previously, the cost of an incorrect specification can be very high, and normally these out-weigh the cost of verification. Manually created proofs have a high rate of errors, even in published proofs [MLP79]. The use of an automated proof checker such as MIZAR-2 [PrR83,PrR88] eliminates human errors in proofs.

1.5. The Specification Language

A formal specification language must be used if a formal verification technique is used. The specification language for protocols must describe the properties of communication protocols including the passage of time and events occurring asynchronously. It must also describe properties which are part of a specification language used for formal verification including sequence of execution, decision making and iteration.

Several specification languages are well known including Estelle [B87], Lotos [Boc86,Boc87,Rud83] and SDL [CCI,CaR82]. The specification language described here, called Protocol Description Language, is similar to Estelle in the description of channel specifications, multiple parties, and types.

Protocol Description Language (PDL) is not as powerful as Estelle since PDL does not have the equivalent of Estelle's "when" clause for specifying channel events. The Estelle "when" clause allows the specification of additional constraints as part of a channel event, such as checking a variable. PDL is much simpler than Estelle in that many language features such as decision making based on the value of variables, functions and procedures are not supported. Estelle has a much richer feature set oriented towards automated implementation rather than formal verification [TsV88].

This specification language is based on an automated protocol specification language developed by Alberta Government Telephones to implement network control protocols. Currently, AGT has applied this protocol specification language to three different protocols used in the control of remote test devices. These protocols provide transfer of commands to remote test devices and transfer of test results back to the controlling computer. The specification language also bears structural similarity to the "Test Manager" developed by IDACOM Electronics Ltd.

1.6. Method of Proof

When safety and liveness are to be proved about a particular protocol the protocol must first be recorded in Protocol Description Language. This description is then translated, by hand, into mathematical statements recorded in the MIZAR-2 language. These statements describe the operation of the protocol in a completely rigorous manner.

A proof of liveness is then written by hand, based on the assumption that the communication links are perfect.

Once the proof of liveness is written, the proof of safety is written, based on the assumption that the communication links are imperfect.

Chapter 2

Survey of Specification and Verification Techniques

2.1. Overview

This chapter briefly describes five different protocol verification techniques. These include two assertion proving techniques similar to the one used here, a state enumeration methodology, a design checking tool based on reachability analysis and a verification technique that utilizes Petri nets. The assertion proving techniques were chosen on the basis that they supported the verification of safety and liveness properties of the protocol studied. The assertion proving verification techniques chosen are SIGETL [TsV88] developed at UBC by Tsiknis and Vuong and Hailpern's Modular Verification [HaO83]. The state enumeration technique is by Blumer and Sidhu [BlS86]. The design checking tool is called VALIRA [VHC86]. The Petri net technique is due to Merlin [Mer79].

One difference between the other assertion proving techniques and that described in this thesis is the use of temporal logic. The two other assertion proving techniques surveyed used temporal logic for specifying and verifying protocols. This chapter also contains a brief description of temporal logic in terms of predicate logic. Since the temporal operators can be expressed in ordinary first order predicate logic, it is also suggested that temporal logic is a notational convenience and does not make one technique significantly different from another, even though it may shorten proofs and simplify the verification process.

2.2. Significant Event Temporal Logic (SIGETL)

The Significant Event Temporal Logic(SIGETL) was developed by George Tsiknis and Son T. Vuong [TsV88].

SIGETL is a method of protocol specification and verification using a temporal logic axiomatic system. That is, the protocol is specified using temporal logic and then specific properties of the protocol are verified using a set of axioms. The method is based on an axiomatic description of each of the modules in a protocol. The axiomatic specification uses the sequence of the events the module has exchanged with the remainder of the system. The sequence of events includes all of the past communications up to the present time. The behavior of the module is based on the events the module has received and is described by temporal logic axioms.

SIGETL is a modified form of Estelle such that formal verification can be performed on the resulting specifications. To this end two major changes are made. First, the behavior is described as temporal logic statements. Specifically, procedure calls are eliminated from the specification. In Estelle transitions are described using a procedural language, much like Pascal. In SIGETL these procedural language statements are described using a predicate notation. Second, SIGETL clarifies two ambiguous areas of Estelle specifications. All modules referenced in the specification must be defined in SIGETL, unlike Estelle. SIGETL also specifies two cases of transitions which are not strictly required in Estelle; a transition is required to specify that only one event occurs at a time and a transition is required to specify what happens when no event occurs. Both of these differences are made to allow formal verification to be performed.

2.2.1. Overview

A significant event sequence is the sequence of events which have been recognized by the protocol party, called a module in SIGETL. A SIGETL module is defined exactly as in Estelle [B87]. An event is the transmission or reception of a message or a time-out. An event is recognized if the module has a transition which corresponds to the event and its enabling conditions are met. Only events recognized by the module become part of the significant event sequence of the module.

Recognized input events cause output events and state transitions to take place. How the module reacts to an input event is based on temporal logic axioms that describe the module.

Channels are described exactly as they are described in Estelle [B87]. Note that only one event can be in a channel at a given time and if, in reality, there is to be a queue of messages in a channel, or some time delay in transmission, this has to be modeled as another module. This constraint is specified by the Directly Coupled Events Rule.

Note that the use of transition axioms makes a SIGETL specification non-deterministic in that specific control-flow is not specified. Instead the next state, modifications to variables and output events are selected by examination of the transition axioms. In practice, as described below, the next state can always be uniquely determined in a SIGETL specification.

2.2.2. Structure of A SIGETL Specification

A SIGETL specification consists of a definition of the types used, followed by the channels used by each party, the abbreviations (notational short-hand conventions) used in the specification and then the list of modules involved.

Each module definition consists of the header, the events which can compose a significant event sequence and the transition axioms. The header describes which channels are used and how they are used. The transition axioms are specified using predicate logic enhanced by operators to describe the notion of the communicating process being in a system state, the recognition of events, and the modifications to the significant event sequence.

The syntax of SIGETL is the same as Estelle except for the specification of transitions. SIGETL simply axiomatizes the semantics of an Estelle transition.

2.2.3. Remarks on the Semantics of SIGETL

The semantics of SIGETL are intended to be like those of Estelle. However, in order to perform a formal verification two areas must be more completely specified.

First, all modules that are referenced in the specification must be defined. These modules are called "Additional Modules". Only safety and liveness properties are specified for these modules. SIGETL requires that the specification be complete. In a SIGETL specification all referenced modules must be defined and it must be explicitly stated that no events can occur at the same time.

The second difference between SIGETL and Estelle is that additional transitions must be specified to cover two cases which are not considered in Estelle. Again, SIGETL must consider these for completeness. A transition is required to specify that only one event occurs at a time. That is, it must be asserted that only one of all the possible events can occur at any specific time. Transitions are also required to specify what happens if an

expected event does not occur. That is, it must be explicitly specified that the protocol does nothing when an event does not occur. Of course, this also allows a specification writer to define that something should happen when no event has occurred.

2.2.4. Verification Using SIGETL

The properties which can be verified in SIGETL are safety and liveness properties for modules and safety and liveness for the system.

In [TsV88] the alternating bit protocol is specified and verified. Two modules are described, SENDER and RECEIVER. Safety properties verified for SENDER are:

- 1) At any time N messages have been received and each has been sent to the network one or more times in the order received from the user.
- 2) If N messages have been sent, $N-1$ have been acknowledged.

Safety properties for RECEIVER are:

- 1) The messages delivered to the user are those received from the network.
- 2) An acknowledgement is sent for each message RECEIVER receives.

Liveness properties for SENDER are:

- 1) The SENDER will continuously send messages to the network as long as the user sends messages to the SENDER.
- 2) Whenever the acknowledgement of a message is received after the message is sent, the acknowledgement is recognized (added to the significant event sequence).

Liveness properties for RECEIVER are:

- 1) If an unbounded number of messages reach RECEIVER then an unbounded number of acknowledgements have been sent.

- 2) If message K does not arrive until the receiver has processed message $K-1$ and messages do not stop coming to the receiver, the receiver will keep sending an acknowledgement for the last message until it receives the next one.

System safety is the following: if N messages have been received by the user attached to RECEIVER, then these are the first N messages sent by the user attached to SENDER and the order has been preserved.

System liveness has two requirements:

- 1) Infinitely many messages are transferred through each one of the four system channels.
- 2) At any time that user2 has received N messages he will receive the $N+1$ st message at some time in the future.

2.3. Modular Verification of Protocols

A technique developed by Brent Hailpern and Susan Owicki, called the Modular Verification technique [HaO83] is described in this section.

This technique applies modular parallel program verification techniques to communication protocols. The modular technique utilizes temporal logic, modular specification and history variables to record sequences of input and output variables.

2.3.1. Overview

This technique is similar to SIGETL in that histories are used to specify input and output variables. These correspond to significant event sequences in SIGETL, although they are not as neatly and thoroughly defined here. As well, while significant event sequences are confined to the description of events, histories can be applied to variables and other properties of the protocol state as well.

Channels in the modular verification technique are specified by axioms and are not constrained to have immediate delivery as specified by SIGETL's directly coupled event rule.

2.3.2. Structure of A Modular Protocol Specification

A modular protocol specification consists of the description of sequential communicating processes, one process for each party in the protocol. The specification of the parties of the protocol is done in a procedural language. Examination of channel variables is used to obtain information about events in the channels. Loops are used to denote non-termination of the protocol while "if" statements specify decision making in the control flow. Operations on the channel modules denote input and output.

2.3.3. Semantics of A Modular Specification

The semantics of a Modular Specification are specified separately from the text of the specification. That is, a specification is hand-translated to temporal logic. This means that an error can be introduced into the specification during the translation from the specification to temporal logic.

The semantics of the operations on the modules are specified by pre-, post- and liveness assertions about each operation which can be performed on the module. As an example, there might be three operations on the channel module: send, receive and existsMessage performing transmission on the channel, reception on the channel and determining if there is a message on the channel, respectively.

By allowing more than one process to access the module, communication between the processes can be specified. The order of events is specified using temporal logic in the specification of the channels.

Hailpern uses both the *henceforth* and *eventually* temporal logic operators in verifications.

2.3.4. Verification of a Modular Specification

The properties which can be verified using modular specifications are exactly the same as can be verified in SIGETL, including safety and liveness for each party and for the system as a whole. This is to be expected since the same proof technology is used, temporal logic. The axiom system is different and different but equivalent temporal logic operations are used [TsV88].

2.4. Verification of Protocol Properties Using State Enumeration

A technique developed by Blumer and Sidhu [BIS86] is described in this section.

This technique involves the enumeration of all system states followed by an analysis to verify completeness, deadlock freeness, livelock freeness, termination and boundedness. The enumeration and analysis processes are automated. For simplicity of presentation and to ensure that this technique is not confused with other techniques it will be referred to as Blumer's method.

2.4.1. Overview

Unlike the other verification techniques that we consider, Blumer's method does not involve algebraic proofs. Instead, the protocol description is mechanically analyzed to determine all of the reachable system states. The resulting state tables are further analyzed to verify the five protocol properties described above.

The specification language of Blumer's method is based on a Pascal [JeW78] subset which has been enhanced for the specification of states and events. The subset of Pascal includes assignment statements, repetition statements, conditionals, procedures, functions, and declarations of constants, types and variables. The Pascal subset has been enhanced so that protocol states, events, event reception and event creation, accessing the parameters of

received events and state transitions.

As in the other protocol specification techniques described here, recognized incoming events cause state transitions. Possible events include timers, received messages from a communication channel and service requests from the client.

Channels are modeled in the protocol specification and can thus be FIFO or non-FIFO. That is, whether the channels can re-order messages can be specified within the protocol specification.

2.4.2. Structure of Protocol Specification Using Blumer's Method

The structure of a specification has four major sections. The first is the types and constants section which is based on the constant and type declaration section of Pascal.

The second section is the interface messages section. This section describes each of the possible messages. The messages are described by naming them and listing the pertinent information for the message.

The third section describes the functions and procedures used in the specification. This section has the header information for functions and procedures using the Pascal syntax. The body of the function or procedure is not supplied. Instead a place holder, the keyword primitive, is shown.

The last section is the specification of the transitions in the protocol. For each transition the state change caused by the message is given as well as an enabling condition which must be true for the transition to occur. This is similar to the "when" clause of Estelle [B87]. Each transition also has a body which describes procedural statements to be executed when the transition occurs.

2.4.3. Semantics of a Specification Using Blumer's Method

The semantics of a specification in Blumer's method is similar to that in other specification techniques. That is, transitions between system states occur upon the reception of recognized events.

Blumer's method is more similar to Modular verification specifications in that procedural programming statements are used to specify changes to state variables. This is also similar to Estelle.

2.4.4. Verification of Protocol Properties Using Blumer's Method

Verification of protocols using Blumer's technique has been automated and for this reason is of interest.

Verification using Blumer's technique involves the consideration of paths through the protocol finite state machines. The state of the protocol is composed of the contents of the transmission and reception channels and the states of the interacting state machines. A protocol makes a transition from one state to another when a message is transmitted or received.

Completeness is verified by ensuring that, for every system state, each event in a channel is received by some transition out of that system state.

Deadlock freeness is verified by checking that all nonfinal system states have at least one possible transition out of the state.

Livelock freeness is verified by ensuring that there are no duplicate paths through the system states.

Termination is ensured if the analyzer completes its path analysis and halts without finding any deadlock states.

Boundedness is checked by halting the analysis if the number of events in a channel is higher than a fixed upper bound.

2.5. Reachability Analysis Using VALIRA

Reachability analysis involves the enumeration of all possible interactions between communicating finite state machines. There are many such systems which generate reachable system states based on a protocol specification [BIS86,Sun79,ZWR80]. VALIRA [VHC86] (VALIdation via Reachability Analysis) is an integrated package which incorporates reachability analysis and a technique to eliminate the "state explosion" problem, prevalent in the application of reachability analysis.

2.5.1. Overview

In a reachability analysis a protocol is defined as two or more communicating finite state machines. Reachability analysis exhaustively generates all reachable global states of the finite state machines. These global states are organized in a reachability tree. Each node in this tree is a global state which represents the contents of all the communication channels as well as the state of each communication finite state machine. The number of states can become prohibitively high, a problem known as state explosion.

VALIRA has two mechanisms to limit state explosion. One technique is to avoid creating duplicate nodes in the tree. That is, if the state already exists, a new state is not created. A second method applies to protocols with unbounded channels. The user may supply channel bounds. He must be careful not to make the bounds too small, so as to lose information, but not so large as to make analysis unwieldy.

2.5.2. Structure of a VALIRA Specification

A VALIRA specification is entered interactively. First, several questions must be answered including the number of processes, whether FIFO channels are used and what the channel is. The state transition arcs are entered for each state machine. The starting state, the ending state, and the type of message is specified for each state transition arc. If the message is transmitted then the message number is negative, on reception the message number is positive.

2.5.3. Analysis Performed by VALIRA

VALIRA performs the following analysis, detection of deadlock nodes, detection of stable states, detection of non-executable interactions and unspecified receptions.

Deadlock nodes are protocol states where no movement can be made. That is, no transmission or reception can occur to move the protocol to a new state.

VALIRA defines stable states as states in which all channels are empty. A state ambiguity exists when a process state appears in more than one stable state. This means that a process of the protocol can have the same channel and process configuration and have it represent two different system configurations. These are not always bad, however "... their semantic intents must be examined with caution [VHC86]."

Non-executable interactions are statements which would not get executed under any conditions. Non-executable interactions are redundant but may indicate the existence of other more serious design errors such as unspecified receptions.

Unspecified receptions are protocol states where a message is available in a channel, but the recipient state machine does not have a state transition arc defined for the message. In this case, the operation of the protocol is undefined.

2.6. Protocol Specification Using Petri Nets

In his 1962 dissertation C.A. Petri [Pet62] used nets to represent the synchronized activities of a parallel automata. Since then these nets have come to be called *Petri nets* and have been applied in many areas [Pet81]. This section describes Merlin's application of Petri nets to the specification of communication protocols [Mer79].

A brief description of the modeling of protocols using Petri nets is taken from [Mer79].

Petri nets model "conditions" represented by *nodes* and "events" represented by *transition bars*. The holding of a condition is represented by placing a token on that node. *Directed arcs* connect nodes to bars and bars to nodes. A transition bar (i.e., event) can fire (i.e., occurs) if all the nodes (i.e., conditions) input to that transition bar have tokens (i.e., hold). When a transition bar fires, it removes one token from each input node and puts one token on each output node.

The tokens are unmarked and are not ordered within the nodes.

2.6.1. Overview

The procedure for specifying a protocol using Petri nets has three steps. First the topology of the protocol must be described by identifying the parties involved and the communication links. Secondly, each of the parties is modeled by a Petri net, independent of the other parties. In the third step the Petri net models of the individual parties are connected to form a global Petri net model of the protocol. On completion, all parties involved in the protocol are specified using one Petri net and the tokens passed between them represent messages exchanged on the communication channels.

Petri nets can specify some protocols which finite state machines cannot. For example, [Mer79] describes the protocol where the sender continuously sends messages to channel which continuously sends them to a receiver. This protocol could not be represented by a finite state machine since the number of tokens in the channel are unbounded and

unordered.

Some protocols with an infinite number of states cannot be represented by a finite state machine. A finite state machine cannot specify a protocol that permits any number of outstanding messages which can be sent and received out of order. This can be modeled by a Petri net in which the number of tokens can grow without limit. The concurrency modeling capability of Petri nets can also be exploited to represent protocols in which several events may occur in arbitrary order. Although concurrency can also be represented by a single finite state machine theoretically, the process is complex and tedious [Mer79].

Petri nets can also specify events that occur in arbitrary order. This is specified by two or more nodes dependent on one bar and a second bar which is in turn dependent on each of these nodes. This type of behavior can be specified in a finite state machine, but the description is not as elegant or concise.

Petri nets are convenient for representing protocols where there is a fixed limit to the number of messages in a channel. The number of tokens put into the system initially could be fixed at a finite number. Limiting the number of tokens at the start could provide a bound to the number of tokens in a channel.

A Petri net could not represent a protocol in which an arbitrary number of outstanding messages are sent and received in order. Petri nets are useful when the messages can be re-ordered by the communication channel. This problem could be addressed using Petri nets with colored tokens.

Another difficulty in using Petri nets is the lack of the concept of time. This deficiency does not permit time-related activities in protocols to be represented. For example, recovery in protocols cannot be elegantly modeled due to the inability to represent timeouts. Merlin [Mer79] has proposed the use of timed Petri nets. However, the analysis of this extended model is more complex and involved.

In general, Petri nets with uniform tokens and without temporal extensions can specify a broader class of protocols than a similarly unadorned state machine notation.

2.6.2. Structure of a Petri Net Specification

A Petri net is a directed graph containing nodes and bars, with a fixed or infinite number of tokens put into the system initially. In Merlin's [Mer79] use of Petri nets, there is no other specification besides that of the Petri net. Merlin does consider the topology of the protocol.

The topology of a protocol is a graph in which the nodes are the parties of the protocol and the arcs are communication links between the parties. The *topology characteristic* of a protocol is the set of topologies the protocol is designed to work on. A protocol can have an *unbounded topology characteristic*, meaning that it has an infinite number of permitted topologies.

2.6.3. Semantics of a Petri Net Protocol Specification

In a Petri net protocol specification, all of the parties are part of one Petri net. Portions of the Petri net may apply to one party or the other. The parties of the protocol communicate by exchanging tokens.

This contrasts with typical state machine specifications where each party is specified by a different state machine. The state machines communicate by exchanging messages over a mutually available, but not necessarily symmetric, communication channel.

2.6.4. Verification Using Petri Nets

Either a state enumeration technique or an assertion proving technique can be used with a Petri net protocol specification. The assertion proving technique, such as SIGETL, Blumer's method or the technique here described, allow safety and liveness to be proved about a Petri net specification.

Petri nets are also amenable to reachability analysis. The verification procedure is similar to that used in a finite state machine specification. The reachability graph is generated from the protocol specification and then the reachability graph is analyzed for specific properties denoted specific qualities of the protocol.

Reachability analysis applied to Petri nets suffers from the same problem when applied to finite state machines, state explosion.

Chapter 3

The Protocol Description Language

3.1. Introduction

This chapter describes the Protocol Description Language(PDL) syntax. The syntax is explained with examples taken from the BNF in Appendix 6 and from the sample specification in Appendix 2.

3.2. Overview of PDL

PDL is a hybrid of finite state machine and programming language description techniques [BoS80]. That is, the parties of the protocol each can have one or more defined states as well as local variables. Each protocol specification can define one or more parties, each of which may communicate with a client using a special channel, and the other parties using one or more channels.

3.3. PDL Syntax

Protocol Description Language is block structured in that each language construct is demarcated by a beginning and an end symbol. All keywords are in lower case and symbols are upper case. For clarity all keywords of PDL will be in bold type-face.

The BNF of PDL is described in Appendix A6.

3.3.1. Protocol Specification Structure

Each specification of a protocol is surrounded by the keywords `protocol` and `end_protocol`. The sections within these keywords are the configuration which defines the channels, and the specification of each of the parties in the protocol. Figure 3.1 sketches the syntax. Note that items within "<" and ">" have to be filled in by the specification writer.

```

protocol <protocol name>
  configuration
    <channel description>
  end_configuration

  party <party name>
    <party definition>
  end_party <party name>

  party <party name>
    <party definition>
  end_party <party name>

  .
  .
  .

  party <party name>
    <party definition>
  end_party <party name>

end_protocol <protocol name>

```

Figure 3.1 Protocol Definition Block

Each protocol specification has one party definition for each party in the protocol. The channel specification describes the channels shared by the parties.

A full specification can be found in Appendix A2 – Example Protocol Description Language Specification.

3.3.2. Channel Specification Structure

The channel description section contains one channel description for each channel used in the specification. The access of each party, if any, to the channel is described as well. Thus, for each channel, the channel name and one or more access lines are described. In each access line the name of the party who has access is given, followed by the type of access, which may be transmit only, receive only and both transmit and receive. Figure 3.2 sketches the syntax of a channel description.

```
channel <name>
    <party name> transmit;
    <party name> transmit-receive;
    <party name> receive;
```

Figure 3.2 Channel Description Section Syntax

Figure 3.3 shows a channel specification taken from the alternating bit protocol.

```
channel MAB  A transmit;
             B receive;
channel MBA  B transmit;
             A receive;
```

Figure 3.3 Channel Description from the Alternating Bit Protocol

This channel specification indicates that A will transmit on channel MAB and B will receive, while on channel MBA the opposite is true.

3.3.3. Party Specification Structure

The party definition has four major parts: the client definition, the local variable description, the initialization section and the description of the states.

Figure 3.4 sketches the syntax for the party definition.

```

party <party name>
  in_client <name series> ;

  out_client <name series> ;

  local <name series> ;

  party_init
    <statement series>
    <next state>
  end_party_init

  <protocol state>

  <protocol state>
  .
  .
  .
  <protocol state>
end_party <party name>

```

Figure 3.4 Party Description

The client definition describes communication between the party and users of the services provided by the protocol or clients. Both `in_client` and `out_client` descriptions are optional so this section may be entirely left out for any party. The specification of a client channel from the alternating bit protocol is

```
in_client SOURCE;
```

This line indicates that channel `SOURCE` will provide input from the client.

The `local` description describes variables used by the party. The variables are visible

only to the party that declares them. Parties can communicate through communication channels and but not through variables.

Variables are not explicitly typed since the only operations that can be applied to them are assignment and transmission to the channels. All variables are variable-length strings. The specification of a local variable from the alternating bit protocol is written as

```
local DATA_TRANSMITTED;
```

Here the local variable DATA_TRANSMITTED is created.

The `party_init` section tells the party what to do when the protocol is started. It may contain a sequence of statements. The `party_init` section must have a `next_state` statement to specify the initial state for the party. The `party_init` section is required. Figure 3.5 shows the specification of a party initialization section from the alternating bit protocol.

```
party_init
  read_client(SOURCE, DATA_TRANSMITTED);

  next_state TRANSMIT_ODD;
end_party_init
```

Figure 3.5 Party Init Section from the Alternating Bit Protocol

Here the party initialization section indicates that when this party is initialized a read will be performed from the client channel SOURCE into the variable DATA_TRANSMITTED and the initial state will be TRANSMIT_ODD. Note that reads from client channels are sequential in that the protocol execution does not continue until it completes.

3.3.4. State Specification Structure

The state definitions define each of the states required for the party. At least one must be defined and as many states as desired can be defined.

The state definition has three sections, the state initialization, the event transition specifications and the unspecified transitions. Figure 3.6 sketches the structure of a state definition.

```

state <name>
  state_init
    <statement series>
  end_state_init

  wait_event

  <event transition>

  <event transition>

  .

  .

  <event transition>

  unspecified
    <statement series>
    <next state option>
end_state <name>

```

Figure 3.6 State Definition

The name of the state is identified by <name> following the state and end_state tokens.

The state initialization section describes what must happen every time the state is entered. Figure 3.7 shows the specification of a state initialization section from the alternating bit protocol.

```
state_init
  transmit "1" || DATA_TRANSMITTED;
end_state_init
```

Figure 3.7 State Initialization Specification

Here the state initialization consists only of the transmission of a message to the output channel. The message consists of a control bit concatenated to the contents of the variable DATA_TRANSMITTED. The two vertical bars symbol " || " is used to denote concatenation.

The wait_event keyword must always follow the state initialization section. It signifies that nothing occurs until an event occurs. Both the state initialization section and the wait_event keyword are required. The event transitions each describe what should occur when a specific event occurs.

The unspecified section describes what should occur when an event occurs which is not specified by one of the event transitions. The unspecified section is required.

The specification of an unspecified section from the alternating bit protocol is

```
unspecified
  next_state TRANSMIT_EVEN
```

This means that when an unspecified event is received the state changes to state TRANSMIT_EVEN.

3.3.5. Event Specification Structure

An event specification has three sections, the event specification, the statements to be executed when the event has occurred and the `next_state` statement. Figure 3.8 sketches the structure of an event specification.

```

receive
    <expression>
    timeout

    <statement series>

    next_state <state name>
end_receive

```

Figure 3.8 Structure of An Event Specification

One of either an expression or timeout must be specified to denote the event that this event transition denotes. The statement series is optional, however the `next_state` statement is required. Figure 3.9 shows the specification of a reception event from the alternating bit protocol.

```

receive
    "ACK1";

    read_client(SOURCE, DATA_TRANSMITTED);
    next_state TRANSMIT_EVEN;
end_receive

```

Figure 3.9 Reception Event Specification

In this event specification, upon the reception of an ACK1, two actions are to be performed. First, a read is to be performed on the client channel SOURCE into the local vari-

able `DATA_TRANSMITTED`. Then the next state is to be set to `TRANSMIT_EVEN`.

3.3.6. Statements

PDL supports the statements to perform the following actions: change states, start and stop timers, transmit data and read and write information to clients.

A `next_state` statement may specify a state name defined by the party or use the keyword `same`. The keyword `same` denotes that the state does not change.

The `start_timer` statement starts a timer which will cause a timeout event to occur at some time in the future. `Stop_timer` stops the timeout event from occurring.

The `transmit` statement causes a message to be transmitted to the output channel.

The `read_client` statement causes a local variable to be assigned the next value from a client's input channel. The `write_client` statement causes a local variable to be written to a client's output channel.

Figure 3.10 is a PDL fragment as an example of the use of the statements.

Figure 3.11 shows a dialogue that this fragment might create.

```
receive
    "wait";

    next_state same;
end_receive

receive
    "go";

    stop_timer;
    next_state CONTINUE;
end_receive

receive
    timeout

    transmit "huh?";
    next_state same;
end_receive
```

Figure 3.10 Example of the Use of Some PDL Statements

```
wait—>

<timeout>

    <—huh?

wait—>

go—>
```

Figure 3.11 Possible Dialogue

Chapter 4

PDL Semantics In MIZAR-2

4.1. Overview

The semantics of PDL are explained with reference to predicate logic. The techniques used to represent time, communication channels, program state, event handling and variables using predicate logic are explained.

The notation used for predicate logic is the MIZAR-2 language. The MIZAR-2 language is briefly introduced.

Finally, the semantics of PDL are explained by describing the translation from PDL to MIZAR-2.

MIZAR-2 [RuD85] is a formal language for the recording of mathematical texts in first order predicate calculus. MIZAR-2 is designed to assist the editing of correct proofs and to allow for automated checking of the proofs. While MIZAR-2 does not prove theorems, it will check proofs. MIZAR-2 has been implemented on a variety of computers and operating systems including Berkeley Software Distribution UNIX[†] 4.3, IBM PC with MS DOS, the Apple Macintosh and IBM system 360. MIZAR-2 was developed at the Institute for Computer Science at the Polish Academy of Science at Warsaw under A. Trybulec.

[†] Registered trademark of AT&T in the USA and other countries.

4.2. Description of MIZAR-2

The following description of MIZAR-2 is taken from [RuD85].

The MIZAR-2 language serves to record reasonings conducted in the first order logic augmented by some notions of set theory. There exists a computer processor for the language which determines whether an input text complies with the MIZAR-2 syntax rules. It contains a checking module which checks whether the recorded inferences are consistent with the rules of logic.

The structure of MIZAR-2 texts is similar to that of mathematical articles. In the first part of such a text, called environment or preliminaries, we display notions and facts assumed to be given. In the second part – called the text proper – we formulate and prove theorems (by hand).

Some MIZAR-2 texts will be explained in detail. Consider the following MIZAR-2 definition.

```
for R, S being Relation pred R <= S
  denotes
for x, y st [x, y] in R holds [x, y] in S;
```

The MIZAR-2 symbol "for" denotes the quantifier "for all". The symbol "being" indicates that the type of the variables R and S is Relation within the context of the statement. Earlier Relation was defined to be an acceptable type to MIZAR-2 within the context of this proof by the statements:

```
given U being nonemptyset;
type Relation denotes subset of [U, U];
```

At this point in the text MIZAR-2 knows nothing about a Relation. Rather, it is aware that a Relation consists of a subset of all the pairs where each element of the pair is a member of a non-empty set.

The MIZAR-2 symbol "pred" indicates that a predicate is being defined. The predicate in this case is an infix binary relation, which accepts two arguments, R and S. The definition of the predicate is defined by another statement which reads

"for all pairs $[x,y]$ in relation R it holds that $[x,y]$ is also in relation S ".

It is important to realize that MIZAR-2 does not know anything about relations. It has a definition that tells it a Relation is a pair of two objects taken from a non-empty set. Later in the text, some predicates on Relations are defined. MIZAR-2 does not know any more about the mathematical concept of relation than is contained in the definition for Relation.

Consider the following MIZAR-2 statement.

```
hence for D being NONNEGATIVE
      ex X,Y being POINT
st DISTANT[X,Y,D] &
  (WHITE[X] & WHITE[Y]) or
  (BLACK[X] & BLACK[Y])
by A;
```

This statement is part of the conclusion of a proof. The first keyword "hence" tells MIZAR-2 that the following statement is part of the conclusion of the proof – a part or the whole of what was to be demonstrated. The next new keyword is "ex" which stands for exists. Thus the first part of the sentence reads

"for all D of type NONNEGATIVE
there exists X,Y of type POINT..."

The remainder of the sentence states a relationship which holds between X,Y and D , described in terms of the predicates DISTANT, WHITE and BLACK. The symbol "&" stands for the logical "and" operation while the keyword "or" stands for logical "or".

The final part of the sentence is the description of the statements that justify this conclusion. These are introduced by the keyword "by". There is only one statement listed, statement "A". A sentence labeled "A" must have appeared previously. There is another statement used in the justification. The keyword "hence" has a two purposes; as well as

introducing the conclusion of the proof, it also means that the previous sentence is also required to justify the conclusion.

A brief description of the structure of a MIZAR-2 text follows in order to explain the use of MIZAR-2 in the verification of PDL specifications.

A MIZAR-2 text contains two sections, the environment section and the proof section. The environment section contains a list of the axioms that are used in the proof section. The environment section can also contain the definition of types and symbols which are used in the axioms and the proofs. The axioms do not require supporting arguments to demonstrate their validity. The proof section contains one or more proofs, each one with a separate list of supporting arguments. Each proof and step within a proof can be labeled so its results can be used later on.

MIZAR-2 texts are block structured and proofs can be nested within each other.

The MIZAR-2 description of the PDL specification makes up the environment section. That is, the specification of the PDL program is the part of the proof text which is accepted without any supporting statements. It is simply a MIZAR-2 description of the PDL program under study.

The consistency and completeness of the axioms in the environment section are not addressed by MIZAR-2. Rather, these must be considered by the person creating the text.

The axioms have to be complete with respect to the needs of the theorems to be proved. In general, if an axiom is not available for a proof one of two situations exist. One of the axioms may have been omitted from the environment section or the theorem to be proved is false. In the case of the proofs described in this paper the first instance means that the translation from PDL to MIZAR-2 is inaccurate. The second situation, that the theorem is not true, means that the PDL specification must be modified.

Inconsistency is another matter. Axioms may be discovered to be inconsistent by inspection. The axioms used in this specification are believed to be consistent as a result of inspection.

4.3. The Alternating Bit Protocol

This section describes the example protocol used in this thesis, the alternating bit protocol as described by Hailpern and Owicki [HaO83]. The entire specification of the alternating bit protocol is given in the protocol description language in Appendix A2 – Alternating Bit Protocol.

The alternating bit protocol consists of two parties, called A and B and two channels, called MBA and MAB. The purpose of the protocol is to transmit data from A to B. The data is read from A's input queue and written to B's output queue.

Parties A and B are connected by two channels, MAB and MBA. MAB can only be used for transmission from A to B, while MBA can only be used for transmission from B to A. That is, A can only write to MAB and read from MBA, while B can only write to MBA and read from MAB.

The channels can only corrupt messages. That is, messages are never lost. The channels guarantee that something will be delivered. A control bit indicates whether the message has been corrupted.

When a message is sent from A to B a flag is sent with it as a sequence flag. First the bit is set to one and then it is set to zero. When B receives a message with the bit set to one, it acknowledges this message with a one. When B receives a message with the bit set to zero, it acknowledges this message with a zero.

If B receives a corrupted message or a message with the bit set incorrectly it transmits

the bit that it was not expecting. So if B receives a zero bit flag when it expected a one bit flag, it would transmit a zero bit flag. If B receives a corrupted message when expecting a one bit flag it would also transmit a zero bit flag.

When A receives a corrupted message or a message that it is not expecting, it retransmits the last message sent, with the same bit flag as it transmitted before.

Notice that when a corrupted or incorrect bit flag is received, exactly the same response is taken, transmit the bit that the protocol party is not expecting. Also, the protocol always waits until one message has been successfully transmitted and acknowledged before transmitting the next message.

4.4. PDL Semantics

This section contains a description of the concepts used to record PDL semantics in MIZAR-2. We hope that the MIZAR-2 formulae are readable without further explanation.

However, the following description, taken from [RuD85], is given to clarify how formal semantics are derived from the PDL specifications.

Each program defines a system

$$M = (S, R)$$

where S is a set of McCarthy state vectors ($\langle \text{control} \rangle$, $\langle \text{data} \rangle$) and R is a next state relation. This system can be characterized by first order axioms.

As an example consider a PDL `next_state` statement:

```
1:next_state NEW_STATE
```

This is essentially a goto from the current position in the program to the first executable statement in the specified state. This statement can be characterized by the following axiom:

for all s, s' (sRs' & $control(s) = 1$) \Rightarrow
 $control(s') = NEW_STATE$ and
 $data(s) = data(s')$.

This means that at control point 1, the control point of the next statement will be that of NEW_STATE and the data (variables) will not change. Assuming that this statement is part of a PDL specification and the state name specified, NEW_STATE , is part of that specification, we can express this fact by the following axiom:

for all s ($control(s) = 1$) \Rightarrow exists s' such that sRs'

A computation of the program M is the sequence of state vectors $s_0, s_1, s_2 \dots$ such that $s_0Rs_1, s_1Rs_2, s_2Rs_3, \dots$ is satisfied. The sequence may terminate, meaning that the last state is not in the domain of R .

Each initial subsequence (i.e. the first k state vectors) of a computation C is called a *history of C*. Given a *history h* containing k state vectors, we define $state(h)$ as s_k and call it *the resulting state of h*.

If there exist s_{k+1} such that $s_0, s_1, s_2, \dots, s_k, s_{k+1}$ is also a history, h' , of C then we say h' follows h and write $h' = Nx(h)$. Nx is an abbreviation for next, referring to the fact that one history immediately follows another.

In PDL semantics we use a modified form of the first description of M ,

$$M' = (S^*, Nx).$$

(S^* is the closure of S , that is, the set of all sequences over S .) The axioms used to describe a PDL specification in terms of the histories of C , their order represented by the relation Nx and the resulting histories.

In the natural semantics of a programming language like Pascal the transition relation

Nx would be a function since the next state is deterministic. Since PDL can specify non-deterministic operations, the transition relation Nx is not, in the general case, a function. While PDL can be used to specify non-deterministic operations this type of protocol is not normally of practical interest. By confining PDL specifications to deterministic operations, the next state relation will always be a function.

Reconsidering our earlier example:

`1:next_state NEW_STATE.`

In the system M' defined above, this statement is now specified as

*for all h such that $control(h) = 1 \Rightarrow$
 $control(state(Nx(h))) = NEW_STATE$ and
 $data(state(Nx(h))) = data(state(h)).$*

4.4.1. Basic Objects

Figure 4.1 lists the communication protocol concepts from PDL which must be recorded in MIZAR-2.

Handling of time.
 Local variables defined in the local section.
 Protocol messages.
 Control points indicating the point of execution.
 Description of communication channels.
 Handling of events.

Figure 4.1 PDL Concepts Recorded in MIZAR-2

The description of each of these concepts relies on how time is handled, since this is central to the notion of a protocol.

4.4.2. Time

Discrete time instants are considered. That is, time occurs in measurable instances which cannot be subdivided further. This simplifies the description of the passage of time which can be modeled as a function returning a discrete value. The actual value of the time unit is irrelevant. We use time only to order events.

Time is modeled as a natural number. The passage of time is denoted by the function $Nx(T)$. $Nx(T) = T + 1$ and in general $Nx\{N\}(T) = T + N$ although this fact is not used in any of the verifications.

Note that if you wish to refer to a point in time that is three time units in the future you would use the notation $Nx\{3\}(T)$.

Shankar [Sha86a] points out that reference to time is not required to prove safety and liveness for data-link protocols. A data-link protocol operates on a link that loses messages but does not duplicate or reorder messages. Time is required for transport protocols, where links can lose, duplicate and reorder messages to an arbitrary extent, with an upper bound on message life times.

In the PDL verifications, time is used as a control point. There is only a loose relationship between this time and physical time. As described above, time is only used to order events.

4.4.3. Local Variables

Rudnicki and Drabent [RuD85] describe the use of variable histories and explicit control points in the formal verification of Pascal programs. In that work the value of the variable at a given point of execution in the program is based on the current history. Thus a variable is modeled as a mapping of a history to a variable's value. Thus the value of a variable is a function of the current history.

The meaning of a variable is modeled as a mapping of a history to a variable's value. Thus, the value of a variable is a function of the time instant considered. Figure 4.2 shows the MIZAR-2 declaration created for each local variable used in the specification.

PDL Definition:

local DATA_TO_BE_TRANSMITTED;

MIZAR-2 Definition:

for T consider DATA_TO_BE_TRANSMITTED being String;

Figure 4.2 MIZAR-2 Declaration of PDL Local Variables

DATA_TO_BE_TRANSMITTED is a local variable in a PDL specification. The MIZAR-2 function DATA_TO_BE_TRANSMITTED maps time to a string. That is, this function determines that value of the PDL variable at any time.

Note that since all variables are variable length strings, all variables are mappings of time to strings.

4.4.4. Protocol Messages

In the example we will consider later, protocol messages are modeled as strings which are concatenations of two other strings, the data and a control bit. The data is an arbitrary string while the control bit can take on a small set of discrete values.

Figure 4.3 shows the MIZAR-2 function definitions for messages.

The function Message maps the data and control bit (both strings) to another string, a message. This function is used to define a message given data and a control bit.

The function Data maps one string, a message, to another string, user data. This function is used to extract data from a message.

```

for D, Bit consider Message being String;
for S consider Data being String;
for S consider ControlBit being String;
MessageDefinition: for D, Bit, S
  st
    Message(D, Bit) = S
  holds
    Data(S) = D &
    ControlBit(S) = Bit;

```

Figure 4.3 MIZAR-2 Definitions for Messages

The function ControlBit maps one string, a message, to another string, a control bit. This function is used to extract the control bit from a message.

The definition MessageDefinition defines the relationship between messages, data and control bits.

Since data is only transferred from one user of the protocol services, across communication channels and back to another user of the protocol service, there is no need to make any further definitions about the nature of the data. Control bits, however, have to have unique, distinguishable values within each channel. Figure 4.4 shows how this fact is recorded in MIZAR-2.

This MIZAR-2 text fragment tells the MIZAR-2 processor that there are six strings with the names, ACK0, ACK1, ZeroString, OneString, EmptyString and CorruptedBit. This statement asserts that ZeroString, OneString and CorruptedBit are one set of distinct values while ACK1, ACK0 and CorruptedBit are another distinct set of values. When a proof must rely on the fact that the control bits are unique within a channel, reference is made to the label StringInequality. Since ZeroString and ACK0 are never in the same channel it is not

```

given ACK0,ACK1,ZeroString,OneString,
    EmptyString,CorruptedBit
    being String such that
StringInequality:ZeroString <> CorruptedBit &
    OneString <> CorruptedBit &
    ACK1 <> CorruptedBit &
    ACK0 <> CorruptedBit &
    ACK0 <> ACK1 &
    OneString <> ZeroString;

```

Figure 4.4 MIZAR-2 Definition of Unique Control Bits

necessary to indicate that they are unique. In fact, it is likely that an implementation of the protocol would use the same value for these two bits.

4.4.5. Control Points

Like all other elements in MIZAR-2 that are not constant over time, control points are histories with respect to time. Since there is a separate control point for each of the communicating parties, the control point function maps time and the name of a party to a control point. Control points are labeled with sequential natural numbers. A party is described sequentially.

Control points are assigned at all executable PDL statements in a specification. This includes next state, start timer, stop timer, transmit, read and write statements.

Figure 4.5 shows the control point function definition in MIZAR-2.

```

for P being Party,T being Time
    consider ControlPoint being natural;

```

Figure 4.5 ControlPoint Definition in MIZAR-2

ControlPoint maps a party and a time to a natural number, which represents a control point.

Each PDL statement which is axiomatized in MIZAR-2 contains a definition of the control point in the current and next time instant. Figure 4.6 shows the control point definition in a MIZAR-2 representation of a transmit statement and (labeled by A2) the use of the transmit statement at control point 2.

```
#define ATransmit(T,D,Bit,NextCP)
  (Full[MAB(Nx(T))]) &
  .
  .
  ControlPoint(A, Nx(T)) = NextCP &
  .
  .
A2:for T st ControlPoint(A, T) = 2
  holds ATransmit(T, DATA_TO_BE_TRANSMITTED(T), OneString, 3);
```

Figure 4.6 PDL Transmit Statement Recorded in MIZAR-2

The first line contains the beginning of the definition of the ATransmit statement. Within the ATransmit definition is the statement that the control point of A in the next time instant will be equal to NextCP (Next Control Point). NextCP is one of the parameters to the definition of the transmit statement.

The transmit statement definition accepts the next control point from the axiomatization of A2, the second control point of party A. The last parameter of the ATransmit predicate is 3, which is the NextCP parameter. Thus the next control point of A2 is 3.

The definition of ATransmit asserts, amongst other things, that the control point at Nx(T) will be NextCP. A2, then, is an axiom that states that any time the control point of

A is 2 at time T, predicate ATransmit will be true with the parameters as specified.

4.4.6. Channels

PDL Channels are modeled by history variables. These variables contain the message held by the channel at any time instant. Thus, to determine whether a channel contains a message at a specific time instant, the history variable is indexed by the time. Thus, the history variable for a channel maps a time instant to a channel.

A channel may be either full or empty, but not both, of course. The contents of the channel is a message.

Figure 4.7 shows the specification that a channel may be either full or empty.

```

for C being Channel pred Full;
for C being Channel pred Empty;
FullOrEmpty:for C being Channel holds
  (Full[C] or Empty[C]) & not (Full[C] & Empty[C]);

```

Figure 4.7 Excluded Middle Definition of Channels in MIZAR-2

Figure 4.8 shows both of the MIZAR-2 channel history variables defined for the channels in the example protocol specification. Note that one of these definitions would be required for each channel in the specification.

```

for T consider MBA being Channel;
for T consider MAB being Channel;

```

Figure 4.8 PDL Communication Channels Recorded in MIZAR-2

The history variables MAB and MBA map time to a channel.

The Contents function maps channels to messages, modeled as strings. Before a message in a channel may be referenced, the contents function must be applied to it. Following is the MIZAR-2 definition of the Contents function.

```

for C being Channel consider Contents being String;

```

Figure 4.9 shows the definition of a transmission to channel MAB.

```

transmit "1" || DATA_TRANSMITTED;

for T, D, Bit st ATransmit[T, D, Bit] holds
  Full[MAB(Nx(T))] &
  (Contents(MAB(Nx(T))) = Message(D, Bit) or
   Contents(MAB(Nx(T))) = Message(D, CorruptedBit))

```

Figure 4.9 Definition of Transmit Statement in MIZAR-2

The first line of the transmit predicate indicates that in the next time instant MAB will be full. The second and third statements indicate that its contents will either be corrupted or not.

The channel modeled by this transmit statement can have only one type of channel failure, to corrupt a message. A message is never lost.

Figure 4.10 shows the definition of a channel that loses messages.

```

transmit "1" || DATA_TRANSMITTED;

for T, D, Bit st ATransmit[T, D, Bit] holds
  Full[MAB(Nx(T))] &
  (Contents(MAB(Nx(T))) = Message(D, Bit) or
  Empty[MAB(Nx(T))]);

```

Figure 4.10 Definition of Transmit Statement in MIZAR-2

This definition defines transmission to channel MAB as resulting in either the channel becoming full in the next time instant, or the channel remaining empty.

4.4.7. Events

There are two forms of events in PDL, reception of messages and timeouts. Message events are messages becoming available in a channel. The state of a channel is determined by the Full and Empty predicates. A transition occurs if the message in the channel matches the transition.

Figure 4.11 shows the transitions generated for the transition on the reception of an ACK1 in state TRANSMIT_ODD in party A of the alternating bit protocol example.

The first line specifies what must occur before the transition occurs, that the contents of the channel must be ACK1 and the control point must be equal to 3. The second line of the implication specifies execution sequencing.

When both of these conditions are true (at control point 3 and the channel contains ACK1), in the next time instant, denoted by $Nx(T)$, the control point will have been advanced to 4 and the channel will be empty.

```

receive
  "ACK1";
for T st ControlPoint(A, T) = 3 &
  ControlBit(Contents(MBA(T))) = ACK1
holds
  ControlPoint(A, Nx(T)) = 4 &
  Empty[MBA(Nx(T))]

```

Figure 4.11 ACK1 Reception and Message Transition in MIZAR-2

4.5. Axiomatization of PDL Statements in MIZAR-2

This section contains a description of how PDL statements are described in MIZAR-2.

4.5.1. Overview

The statements of PDL are axiomatized (a single general purpose MIZAR-2 axiom for each PDL statement) so that as little repetition as possible is required when defining a protocol specification. However, the axiomatization is not sufficiently general so that it could be applied to any PDL specification.

Most statements are axiomatized with one MIZAR-2 predicate, though some require more. The statements are grouped together in control points so that more than one statement is combined together in a single axiom.

The axiomatization must indicate how each statement changes the state of execution and that everything else does not change.

4.5.2. Selection of Control Points

In general, the fewer the control points, the shorter and less tedious is the proof. However, one cannot combine statements into control points unless the order of the execution of the statements does not matter. This problem is similar to the problem that occurs in proofs of sequential programs.

Figure 4.12 shows three Pascal language statements and their axioms that cannot be combined into a single control point. Note that this figure assumes the Pascal axiomatization is similar to that used for PDL.

```

a := 0; { Control Point 0 }
a := 1; { Control Point 1 }
b := a; { Control Point 2 }

for T st ControlPoint(T) = 0 holds
    a(Nx(T)) = 0 &
== Specification that there were no other changes.
    .
    .
    .

for T st ControlPoint(T) = 1 holds
    a(Nx(T)) = 1 &
== Specification that there were no other changes.
    .
    .
    .

for T st ControlPoint(T) = 2 holds
    b(Nx(T)) = a(T) &
== Specification that there were no other changes.
    .
    .
    .

```

Figure 4.12 Three Pascal Statements And Their Axiomatizations

If these three axioms were combined in a straightforward way, the effect would be to specify that variable `b` would be assigned the value of 0 rather than the value of 1.

PDL statements should be combined into control points unless there is order of execution dependent code, such as assignments to variables or transmissions of messages or if a `wait_event` statement is encountered.

4.5.3. Sequencing Control

Each MIZAR-2 axiom describing a control point must specify the next control point. In general, the statement axioms contain a parameter which specifies the next control point. Any jumps in execution due to `next_state` or transition execution are handled by changing the next control point parameter.

4.5.4. Generation of Axioms for Each Statement

This section will describe how each PDL statement is axiomatized. The channel configuration, declaration of client channels and local variables have already been described. Thus, only executable statements will be considered.

4.5.4.1. Read_Client

The `read_client` statement causes information to be taken from a client channel and transferred into local variables. Figure 4.13 shows a PDL `read_client` statement and the corresponding MIZAR-2 axioms that describe it.

Note that this axiom is specific in that it can only be used with party A to read from channel `SOURCE` into variable `DATA_TO_BE_TRANSMITTED`.

The first line indicates that at the next time instant local variable `DATA_TO_BE_TRANSMITTED` will equal the value of channel `SOURCE` at position `SOURCE_length`. The second line indicates that the `SOURCE_length` will be increased by

```

Read_Client(SOURCE, DATA_TO_BE_TRANSMITTED);

#define Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED(T,NextCP)
  DATA_TO_BE_TRANSMITTED(Nx(T)) = SOURCE(SOURCE_length(Nx(T))) &
  SOURCE_length(T) + 1 = SOURCE_length(Nx(T)) &
  ControlPoint(A, Nx(T)) = NextCP &
  AChannelsUnchanged(T)

for T st ControlPoint(A, T) = 1
  holds Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED(T, 2);

```

Figure 4.13 Axiomatization of read_client

one so that the next read on the channel will obtain the next message. The third line indicates that the control point will be changed. The Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED axiom has two parameters, one is the time instant (T) and the second is the next control point (NextCP). The parameter NextCP specifies the control point at the next time instant. In this example, that would be 2. The last line indicates that the channels used by A will not be changed.

A1 states that when the control point of A is 1, the predicate Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED is true with the parameters being the current time and control point 2.

4.5.4.2. Next_State

The next_state statement causes the current state to be changed to the state specified. This causes the control flow to jump to the state initialization section of the specified state, not unlike a goto statement. There is no explicit axiomatization of the next_state statement since it can always be combined with another statement, except another next_state statement. However, it is not sensible to have two next_state statements in sequence since the second one would never be reached.

4.5.4.3. Transmit

The transmit statement causes a message to be transmitted to the channel specified, and it defaults to the channel the party has transmit access to.

Figure 4.14 shows how a transmit statement is axiomatized.

```
#define ATransmit(T,D,Bit,NextCP)
  (Full[MAB(Nx(T))] &
   (Contents(MAB(Nx(T))) = Message(D,Bit) or
    Contents(MAB(Nx(T))) = Message(D,CorruptedBit)) &
   ControlPoint(A, Nx(T)) = NextCP &
   MBA_For_A_Unchanged(T) &
   ALocalsUnchanged(T)
```

Figure 4.14 Axiomatization of Transmit

Note that this axiom is specific in that it can only be used with party A and channel MAB.

The first line indicates that at the next time instant channel MAB will be full. The second line and third lines indicate that at the next time instant the contents of MAB will be equal to the message formed from data "D" and control bit "Bit" or a corrupted message. The fourth line indicates that the control point will change to the parameter passed to the axiom. As with other axiomatizations of PDL statements discussed, this parameter is called NextCP. The fifth line indicates that at the next time instance, predicate MBA_For_A_Unchanged will be true while the last line indicates that none of the local variables of A will change.

The transmit statement models the channel's reliability. In the protocol considered here, the alternating bit protocol, the message can either be delivered corrupted or uncorrupted. The message is never lost.

The predicate `MBA_For_A_Unchanged` indicates that party A will not change the channel MBA. In this protocol MBA is the channel to which party A has receive access. There is another predicate `MAB_For_B_Unchanged` which indicates party B will not change the channel MAB. Figure 4.15 shows the definitions of `MBA_For_A_Unchanged` and `MAB_For_B_Unchanged`.

```
#define MBA_For_A_Unchanged(T)
    Full[MBA(T)] implies
        (Full[MBA(Nx(T))] &
         (Contents(MBA(T)) = Contents(MBA(Nx(T)))))

#define MAB_For_B_Unchanged(T)
    Full[MAB(T)] implies
        (Full[MAB(Nx(T))] &
         (Contents(MAB(T)) = Contents(MAB(Nx(T)))))
```

Figure 4.15 `MBA_For_A_Unchanged` and `MAB_For_B_Unchanged`

`MBA_For_A_Unchanged` indicates that party A will not change the channel to which it has receive access. It states that if the channel is full it will remain full and the contents will remain unchanged. If the channel is empty, its state cannot be determined by examination of the execution of party A, since it could be filled by party B in this time instant.

Similarly, `MAB_For_B_Unchanged` indicates that party B will not change the channel to which it has receive access. If the channel is full it will remain full and the contents will remain unchanged. If it is empty it could be filled by party A in this time instant.

4.5.4.4. Write_Client

The `write_client` statement causes information to be transferred from local variables into a client channel Figure 4.16 shows the MIZAR-2 axiom that describes a `write_client` statement.

```
#define Write_Client_SINK_DATA_RECEIVED(T, NextCP)
  DATA_RECEIVED(T) = SINK(Nx(T), SINK_length(T)) &
  (SINK_length(Nx(T)) = SINK_length(T) + 1) &
  ControlPoint(B, Nx(T)) = NextCP
```

Figure 4.16 Axiomatization of `write_client`

Note that this axiom is specific in that it can only be used with party B to write to channel SINK from variable DATA_RECEIVED.

The first line indicates that at the next time instant channel SINK at position SINK_length will be equal to local variable DATA_TO_BE_TRANSMITTED at time T. The second line indicates that the SINK_length will be increased by one so that the next read on the channel will obtain the next message. The third line indicates that the control point will be advanced to that specified by the parameter NextCP.

4.5.5. MIZAR-2 Text For Wait_Event Statement

The `wait_event` command indicates that the party will wait until one of its defined events occurs. The MIZAR-2 text which describe the `wait_event` command must also represent all of the transition specified, the `unspecified` statement and what occurs when no event occurs.

Figure 4.17 shows the PDL statements used in the examples in this section.

```
wait_event;  
    receive  
        "ACK1";  
    .  
    .  
    .  
end_receive  
unspecified  
    next_state TRANSMIT_ODD;
```

Figure 4.17 Wait_Event Example

The control point associated with the `wait_event` has one axiom for each transition specified, one for the unspecified transition and one axiom to specify what should take place when no events have occurred. All of these axioms could be combined into one axiom, but leaving them separate makes writing proofs simpler and shorter.

The axiom which defines what occurs when no event has occurred states that, at any time when the control point is at the `wait_event`, and no timeout has occurred and the receive channel is empty, the control point at the next time instant does not change.

Figure 4.18 shows the axiom used when no event has occurred.

```

A3x3:for T st ControlPoint(A, T)= 3 & Empty[MBA(T)]
    holds ControlPoint(A, Nx(T)) = 3 &
    ALocalsUnchanged(T) &
    AChannelsUnchanged(T);

```

Figure 4.18 Axiomatization of No Event Occurring

This axiom is for control point 3, and happens to be the third of the three wait_event axioms required for control point three.

Axiom A3x3, which defines what occurs when a defined transition occurs, states that, at any time when the control point is at the wait_event and the receive channel contains the desired message, the control point at the next time instant advances and the channel becomes empty, while the transmit channel does not change. Figure 4.19 shows an event transition.

```

A3x1:for T st ControlPoint(A, T)= 3
    & ControlBit(Contents(MBA(T))) = ACK1
    holds ControlPoint(A, Nx(T)) = 4 &
    Empty[MBA(Nx(T))] & MAB_For_A_Unchanged(T) &
    Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED(T, 4);

```

Figure 4.19 Axiomatization of Transition on Event Reception

Axiom A3x1 is for control point 3, and happens to be the second of the three wait_event axioms required for control point three. Note that this control point also contains a read_client statement.

Another axiom, in this case A3x2, defines what happens when an undefined transition

occurs. It states that, at any time when the control point is at the `wait_event`, and the receive channel contains a message that has not been specified by another transition, the control point at the next time instant advances to the unspecified section and the channel becomes empty, while the transmit channel does not change. Figure 4.20 shows axiom A3x2 which defines the unspecified function.

```
A3x2: for T st ControlPoint(A, T)=3
    & ControlBit(Contents(MBA(T))) <> ACK1
    holds ControlPoint(A, Nx(T)) = 2 &
    Empty[MBA(Nx(T))] &
    ALocalsUnchanged(T, MAB_For_A_Unchanged(T);
```

Figure 4.20 Transition on Unspecified Event Reception

Chapter 5

Verification Procedure

5.1. Overview

This chapter describes how safety and liveness are verified for Protocol Description Language specifications. This chapter also contains a discussion of our claim that by proving safety and liveness for perfect channels and proving safety for imperfect channels, that safety and liveness for imperfect channels can be concluded.

5.2. On the Use of Temporal Logic

Temporal logic is very natural for use with communication protocols since the notion of time is basic to many protocols. For example, timeouts are used in all commonly used protocols such as HDLC, SDLC, Bisync and collision type LAN protocols.

Temporal operators commonly used are "henceforth", "eventually" and "until". Hailpern [HaO83] utilizes the "henceforth" and "eventually" operators. Tsiknis and Vuong [TsV88] use only the "until" operator. Any of these are equivalent since they can all be expressed in terms of each other. Figure 5.1 shows these relationships.

$$\text{henceforth } A \iff \neg(\text{eventually } \neg A)$$
$$\text{henceforth } A \iff A \text{ until false}$$

Figure 5.1 Relationship Between Three Temporal Operators

These operations can also be expressed by using quantified time variables. The henceforth operator makes a statement about all time instants in the future. Thus, all values of T greater than T_0 (current time) are considered. The "eventually" operator asserts the

5.3.1. Safety and Liveness Revisited

In chapter 1 safety was described as "bad things will not happen". Safety is the notion that the protocol will not cause an incorrect delivery of information even if there is a failure in a communication channel. One reason for the use of protocols is to provide a service in the face of communication channel failures. Thus, when a channel loses or corrupts a message, the protocol must detect the error and recover, eventually delivering the information.

If a channel fails and never again provides proper delivery of a message then it is obvious that the protocol cannot recover to the point of delivering the information.

Liveness means that the services provided by the protocol are implemented. In the case of the alternating bit protocol, the service is transfer data from party A to party B. The alternating protocol is live if data messages are transferred from A to B.

A protocol is safe if, under all possible channel failures, the protocol does not lose messages or deliver messages it should not without detecting the error. For the alternating bit protocol we assume that the channel only detectably corrupts the messages.

5.3.2. A Protocol's System State

The system state of a protocol is determined by the value of all the quantities that could change. This includes the control point for each party, the values of the local variables, whether there are timers pending and the contents of the channels. The system state of a protocol can change even if the states of both parties have not changed, for example, when a channel's contents change. The execution of PDL statements causes the system state to change in a deterministic way, except in the case of channel failures. When a transmission statement occurs, the channel may or may not fail. Thus, in this case, the system will not change in a deterministic way.

Note that if a local variable's content are not examined or used except to transfer data between a client channel and communication channels, it does not meaningfully affect the system state. Figure 5.4 describes the elements that make up the system state in the PDL specification for the alternating bit protocol.

State of each party
 Control point of each party
 Contents of all channels
 State of timers

Figure 5.4 Elements of the System State

5.3.3. Protocol Procedures

As a protocol goes through its procedures, its system state changes. If the protocol has a finite number of states and it does not terminate, it follows that the protocol will be in some system states more than once or it will remain in one state for an infinite amount of time. In general, the protocol's procedures define cyclic paths through the system states.

The procedures of a protocol which provide delivery of information can be called the liveness procedures, while those that provide recovery from channel failures can be called safety procedures.

The liveness procedure in the alternating bit protocol provide for the delivery of messages from party A to party B. Party A sends a message with the control bit set to one and then receives an ACK1 from party B. Party A then sends a message with the control bit set to zero and receives and ACK0 from party B. This cycle of four message is then repeated.

The safety procedures in the alternating bit protocol have to recover from any message being corrupted in the channel. There are four different messages that are transmitted and a safety procedure to recover from the incorrect transmission of each. In each case the

previous message is re-transmitted.

For example, the normal exchange of messages with the signaling bit set to zero

<message, 0> —>

<— <ACK0>

If the message is corrupted once then the sequence of messages is

<message, 0> —> (corrupted)

<— <ACK1>

<message, 0> —>

<— <ACK0>

In this case the transmission of ACK1 and the re-transmission of the message with the bit set to zero is the safety procedure. Four safety procedures handle all lost messages.

The safety procedures are utilized only when an error is detected and recovery is taking place. The liveness procedures are completed only when the channels have provided correct delivery of all messages required for the information transfer.

The protocol system states and state transitions can be considered nodes and edges in a labeled, directed graph. The total number of system states is the product of the number of different values each of the quantities which affect the system state. In general, the reachable states which are the combinations that could actually occur, are much fewer than the total number of combinations. There are many such systems which generate reachable system states based on a protocol specification [BIS86,Sun79,ZWR80] as described in chapter 2.

For the alternating bit example there are four combinations of party states and control points and for each of these there are two combinations of channel contents. In all, there are eight system states of interest. Figure 5.5 shows the reachable system state graph, generated by hand, for the alternating bit protocol example.

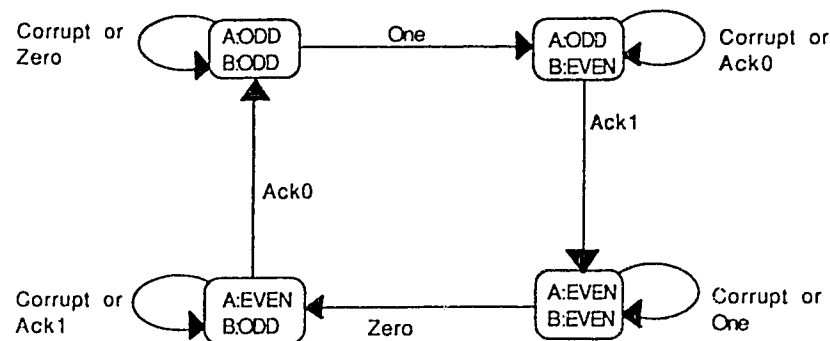


Figure 5.5 System State Transition Graph

The transitions associated with the safety procedures are those caused by the reception of a corrupted message. Notice that each of these procedures results in a return to the original system state once the message is correctly transferred. What occurs is that a corrupted message causes the safety procedures to be invoked, which, when the channel has transmitted the corrupted message correctly, allow the liveness procedures to continue.

The proof of liveness requires us to show that the protocol will cycle through the liveness procedures, successfully delivering information. The proof of safety requires us to show that the safety procedures always causes a return to the same system state in which the message was received, so that the liveness procedures can continue.

Thus, one can prove liveness, without reference to the safety procedures, and safety without reference to the liveness procedures, except that the safety procedures return to the system state before the error in transmission occurred.

Since safety procedures are only required for protocols using imperfect channels, it is unnecessary to prove safety for a protocol using perfect channels. Since liveness does not require reference to channel failures, it is unnecessary to consider channel failures when

proving liveness. It is important to note that this conclusion only applies when the system state transition graph indicates that the safety procedures always return the protocol to the system state that the safety procedures began in. When a protocol has this quality we call it stable.

5.4. Modular Proof Structure

This section describes how the proofs for safety and liveness can be modularized in MIZAR-2.

5.4.1. Lemmas in MIZAR-2

MIZAR-2 allows a proof to be made by first proving a sequence of lemmas. The lemmas are referenced in the proof by referring to the label given to the lemma. This technique is extremely useful if you can re-use the lemmas or if the proof is very large. This is the case in the proofs of safety and liveness for communication protocols.

One lemma is done to describe each of the state transitions. This section describes these lemmas.

5.4.2. Lemma Structure

Each lemma describes one state transition. Therefore, the assumptions for the lemma must describe the system state before the transition and the conclusion of the proof must describe the system state after the transition. The elements in the system state are described in section 4.2.2. All elements of the system state must be described in the assumptions and conclusions of the proof, even if they do not change.

The assumptions and conclusions section of the lemmas are analogous to pre- and post- assertions used in other proof systems and specification techniques.

The proof body of the lemma consists of using the MIZAR-2 description of the protocol specification to prove that the conclusions can be derived from the assumptions.

5.5. Verification of Safety

The verification of safety consists of proving that the protocol is stable. To prove the protocol is stable, we must prove that each of the system states that the liveness procedures traverse is stable.

A system state is stable if for all possible channel errors, the protocol does not leave that system state, until the protocol has recovered from the channel error.

To create the safety proof, a separate proof must be created for each of the reachable system states. This proof must show that for all possible channel errors the safety procedures are executed instead of the liveness procedure advancing. This proof must also show that the protocol will eventually return to the state in which the error occurred. Should the error re-occur the same safety procedure will be executed since the protocol is in exactly the same state as when the first error occurred.

Figure 5.6 shows the steps required for the safety proof.

5.6. Verification of Liveness

The verification of liveness consists of proving that the protocol performs useful work when the channels do not fail. The definition of useful work depends on the services provided by the protocol but in general consists of the transportation of some information over media. Liveness must be defined specific to the protocol under consideration.

Normally the liveness properties are described inductively because the protocol iteratively transfers information. That is, the definition of liveness states that assuming the liveness property is true at some time, it will be true at some time in the future and some use-

-
- 1) Enumerate all reachable system states.
 - 2) For each reachable system state enumerate all possible inputs beside those part of the liveness procedures.
 - 3) For each system state prove that the protocol system, upon receipt of an unexpected input, will eventually return to the system state in which the corrupted message was received, and the system will be ready to accept another message.
-

Figure 5.6 Steps of a Safety Proof

ful work will have been done.

The protocol's purpose may be to transfer information over the communication media from an input queue to an output queue, with the output queue growing as a prefix of the input queue.

In general to prove liveness, liveness must be formally defined. This definition must define useful work as changes to an output queue or some other time-dependent state variable.

To actually perform the liveness proof the first step is proving that the liveness property can be true at time 0, establishing the basis for the inductive proof. One must then assume the liveness property is true at a point in the liveness procedure cycle. Then, using the definition of the protocol, one must prove that the protocol goes through the liveness procedure and returns to the point at which the liveness property was assumed to be true. In the process one must prove that useful work was performed, as defined by the liveness definition.

5.7. Verification of Safety in the Alternating Bit Protocol

The verification of safety in the alternating bit protocol consists of proving that the protocol is stable. To prove the protocol is stable we must prove that each of the system states that the liveness procedures traverse is stable.

A system state is stable if two conditions are met. First, transitions caused by the receipt of corrupted messages do not cause the protocol state to be advanced along the liveness state. Second, the transition taken on receipt of the corrupted message will eventually result in the protocol returning to the state in which the error occurred.

Each of the transitions caused by the receipt of corrupted messages cause the protocol to return to the originating state, possibly via an intermediate state. This is proved in four separate lemmas, one for each of the liveness transitions. In each of the lemmas it is proved that the protocol will return to the originating state within two transmissions.

Each of the lemmas describes the values of all of the system state variables before and after the transition.

Appendix A4 Proof of Safety Procedures in MIZAR-2 contains all four lemmas required for the proof of safety for the alternating bit protocol.

The structure of each of the four lemmas is similar. The assumptions section defines the values of all of the system variables before the transition has occurred. The value of the message in the channel is known to be corrupted. The lengths of the input and output queues are assumed to have arbitrary fixed values. The other channel is assumed to be empty. Each of the lemmas describes the transmission of two messages, one by each of the parties.

The lemmas each have three sections. The first section describes the control point of one of the parties advancing until a response transmission has been made to the erroneously

transmitted message. The second section of the lemma describes the control point of the second party when it retransmits the message while the third section describes the response of the initial party as it accepts the newly transmitted message and returns to the original system state which is part of the liveness procedures.

Note that it is a quality of the alternating bit protocol that it returns to a system state in the liveness procedure within two transmissions after the erroneous transmission even if the transmissions made during recovery are erroneous.

Figure 5.7 shows the assumptions and conclusion of the safety transition lemma recorded in MIZAR-2. This transition covers the case when a message with the bit set to one is corrupted.

```

Transition1Corrupt:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 3 &
    Contents(MAB(T)) = Message(SOURCE(N), CorruptedBit) &
    SOURCE_length(T) = N & SINK_length(T) = M &
    ControlPoint(B, T) = 1 & Empty[MBA(T)];

    .
    .
    .
  hence ex T' st
    ControlPoint(B, T') = 1 &
    Empty[MBA(T')] & SINK_length(T') = M &
    (Contents(MAB(T'))
      = Message(DATA_TO_BE_TRANSMITTED(T'),OneString) or
    Contents(MAB(T'))
      = Message(DATA_TO_BE_TRANSMITTED(T'),CorruptedBit)) &
    ControlPoint(A, T') = 3 & SOURCE_length(T') = N by Part2;
end;

```

Figure 5.7 Safety Transition Lemma in MIZAR-2

These assumptions and conclusions show that after another message is transmitted that the control points will be the same and the lengths of the input and output queues will not have changed.

Figure 5.8 shows the assumptions and conclusion of the safety transition lemma recorded in MIZAR-2. This transition covers the case when a message with the bit set to one is corrupted.

```

ControlBit(Contents(MAB(T))) <> OneString
  by Assumptions, StringInequality, MessageDefinition;
then S1:ControlPoint(B, Nx(T)) = 3 &
  SINK_length(Nx(T)) = M &
  Empty[MBA(Nx(T))] &
  Empty[MAB(Nx(T))] by B1x2, Assumptions;
then S1':ControlPoint(B, Nx{2}(T)) = 1 &
  (Contents(MBA(Nx{2}(T)))
    = Message(EmptyString, ACK0) or
    Contents(MBA(Nx{2}(T)))
    = Message(EmptyString, CorruptedBit)) &
  SINK_length(Nx{2}(T)) = M by B3;
then ControlBit(Contents(MBA(Nx{2}(T)))) <> ACK1
  by StringInequality, MessageDefinition;
then Part1:ControlBit(Contents(MBA(Nx{2}(T)))) <> ACK1 &
  ControlPoint(B, Nx{2}(T)) = 1 &
  SINK_length(Nx{2}(T)) = M by S1';

```

Figure 5.8 Safety Transition Response of Party B in MIZAR-2

This section of the lemma proves that B will respond to the corrupted bit by sending an ACK0. Statement S1 concludes that B will empty the channel MAB and nothing will have been put into channel MBA. Statement S1' concludes that B will transmit an ACK0 on channel MBA that will arrive uncorrupted or corrupted. Statement Part1 concludes that party B has filled the channel MBA with a message that is not an ACK1.

Figure 5.9 contains the proof that party A will respond to the ACK0 by retransmitting

the same data and the same control bit. Statements S2 and S3 concludes that the control point of A will not change for two time instants. Statement S4 shows that A will then remove the message from the channel MBA. Statement Part2 shows that A will then retransmit the same message with the same control bit as previously, a one.

```

S2:ControlPoint(A, Nx(T)) = 3 &
  SOURCE_length(Nx(T)) = N by Assumptions, A3x3;
then S3:ControlPoint(A, Nx{2}(T)) = 3 &
  Empty[MAB(Nx{2}(T))] & SOURCE_length(Nx{2}(T)) = N
  by A3x3, S1;
then S4:ControlPoint(A, Nx{3}(T)) = 2 &
  SOURCE_length(Nx{3}(T)) = N &
  Empty[MBA(Nx{3}(T))] &
  Empty[MAB(Nx{3}(T))] by A3x2, Part1;
then Part2:ControlPoint(A, Nx{4}(T)) = 3 &
  (Contents(MAB(Nx{4}(T))) =
    Message(DATA_TO_BE_TRANSMITTED(Nx{4}(T)), OneString) or
    Contents(MAB(Nx{4}(T))) =
      Message(DATA_TO_BE_TRANSMITTED(Nx{4}(T)),
        CorruptedBit)) &
  SOURCE_length(Nx{4}(T)) = N by A2;

```

Figure 5.9 Party A Safety Response in MIZAR-2

Figure 5.10 proves that the control point of party B does not change while A is retransmitting. That is, party B will wait on channel MBA while A performs its the PDL statements required to transmit another message. The wait_event axiom which specifies the wait for the message is Blx3.

```

ControlPoint(B, Nx{3}(T)) = 1 & SINK_length(Nx{3}(T)) = M
  by Part1, B1x3, S3;
then ControlPoint(B, Nx{4}(T)) = 1 &
  Empty[MBA(Nx{4}(T))] &
  SINK_length(Nx{4}(T)) = M by B1x3, S4;

```

Figure 5.10 Conclusion of Party B Safety Response in MIZAR-2

5.8. Verification of Liveness in the Alternating Bit Protocol

To prove that the alternating bit protocol is live we must prove that the output queue is a prefix of the input queue and that it grows over time. The PrefixInduction axiom formally defines this concept.

This section describes the theorem which must be proved to demonstrate liveness. In the proof text, this theorem is called PrefixInduction and the predicate is called prefix.

To prove liveness in the case of the alternating bit protocol, one must prove that information is being transferred from one client to the other. The input client channel, SOURCE, is modeled as an infinite array, where each member is accessible. The output client channel, SINK, is modeled as an unbounded array. Given these definitions, liveness means that the output client channel is a prefix of the input client channel. Figure 5.11 shows the definition of prefix.

```

for N,T being natural pred prefix;
PrefixInduction:for N,T,T'
  st
    prefix[N,T] &
    SINK(T', N + 1) = SOURCE(N + 1) &
    SINK(T', ((N + 1) + 1)) = SOURCE((N + 1) + 1)
  holds
    prefix[((N + 1) + 1), T'];

```

Figure 5.11 Definition of Predicate Prefix

Axiom PrefixInduction states that if the conditions specified by Prefix exists at time T, of length N and that at time T' the N + 1st and N + 2nd messages are the same in both the input and output queues (i.e. two messages have been properly transferred), then a prefix of length N + 2 exists. The induction axiom describes the transfer of two messages

because the liveness procedure in the alternating bit protocol transfers two messages in each liveness procedure cycle.

The first step in the proof of liveness is the basis step. The basis step involves proving that for a time T when the initial protocol conditions are true and there is prefix of length zero there exists a time T' such that there is a prefix of length one and the protocol conditions are exactly those described in the inductive step.

The proof of the prefix predicate is based on four lemmas. These lemmas are then used to prove the existence of four times at which the contents of system state variables are known. The existence of these four times, with corresponding information about the state variables are used to directly prove the PrefixInduction theorem about the alternating bit protocol.

Figure 5.12 shows the state transition diagram with the lemma names beside the transitions they refer to.

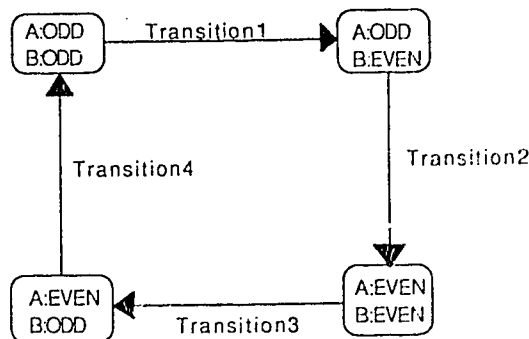


Figure 5.12 General Structure of Liveness Proof

Each of the four lemmas describes one of the transitions through the protocols liveness

procedure. Each lemma describes the reception of one message by one party and the transmission of a response message by the other party. The assumptions and conclusions for each of the lemmas describes the system state variables before and after the transmission. The assumptions of each of the lemmas follows directly from the conclusion of another of the lemmas.

Lemmas Transition1 and Transition3 describe the reception of a message by party B from party A. Party B then responds by transmitting an acknowledgement. Lemmas Transition2 and Transition4 describe the reception of the appropriate acknowledgement message by party A, sent by party B.

All four lemmas have two sections to the proof and proceed in a similar manner. First, the change in state of the receiving party is proved followed by a proof of the change in the state of the transmitting party.

Once the four lemmas are proved these are used to prove the existence of the four interesting control points using the MIZAR-2 "consider" statement. The qualities of these times are used to prove the PrefixInduction theorem, completing the proof of liveness.

Figure 5.13 shows the assumptions and conclusion for the lemma transition 1. This lemma concludes that if party A sent a message with the control bit set to one then B will acknowledge that message with an ACK1. Figure 5.14 shows that party B will respond to the message by putting it in the output queue and sending an ACK1 in return. Statement S1 concludes that B will remove the message from the channel. Statement Part1 concludes that B will transmit an ACK1.

Statement S1 asserts that both the channels MBA and MAB are empty. The state of channel MBA can be deduced from statement Blx1 while the state of channel MAB can be concluded directly from the assumptions. Note that the fact that axiom Blx1 is applicable is deduced from the assumptions. Note that value of SOURCE is equated directly to SINK,

```

Transition1:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 3 &
    Contents(MAB(T)) = Message(SOURCE(N), OneString) &
    SOURCE_length(T) = N &
    SINK_length(T) = M &
    ControlPoint(B, T) = 1 &
    Empty[MBA(T)];
    .
    .
    .
  hence ex T' st
    ControlPoint(B, T') = 4 &
    SOURCE(N) = SINK(T', M + 1) &
    Contents(MBA(T')) = Message(EmptyString, ACK1) &
    SINK_length(T') = M + 1 &
    SOURCE_length(T') = N &
    ControlPoint(A, T') = 3 &
    Empty[MAB(T')]
    by Part2;
end;

```

Figure 5.13 Liveness Transition Assumptions and Conclusion in MIZAR-2

that is the input to the output. This is required since later on the proof of liveness depends directly on the relation between these two queues.

Statement Part1 concludes that B will transmit an ACK1, but also carries the equation between SINK and SOURCE on from statement S1.

Figure 5.15 shows how A will remove the ACK1 from the channel MAB. This involves asserting that the channel MAB will be empty in the next state. The other state changes described in this fragment indicate that the length of the input queue, SOURCE, will not change.

```

ControlBit(Contents(MAB(T))) = OneString
  by Assumptions, MessageDefinition;
then S1:ControlPoint(B, Nx(T)) = 2 &
  SOURCE(N) = SINK(Nx(T), SINK_length(T)) &
  (SINK_length(Nx(T)) = SINK_length(T) + 1) &
  Empty[MBA(Nx(T))] &
  Empty[MAB(Nx(T))] by B1x1, Assumptions;
then Part1:ControlPoint(B, Nx{2}(T)) = 4 &
  SOURCE(N) = SINK(Nx{2}(T), SINK_length(Nx{2}(T))) &
  Contents(MBA(Nx{2}(T))) = Message(EmptyString, ACK1) &
  SINK_length(Nx{2}(T)) = M + 1 by B2, Assumptions;

```

Figure 5.14 Response of B in Liveness Procedure in MIZAR-2

```

ControlPoint(A, Nx(T)) = 3 & SOURCE_length(Nx(T)) = N
  by Assumptions, A3x3;
then ControlPoint(A, Nx{2}(T)) = 3 &
  SOURCE_length(Nx{2}(T)) = N &
  Empty[MAB(Nx{2}(T))]
  by A3x3, S1;

```

Figure 5.15 Response of A in Liveness Procedure in MIZAR-2

Chapter 6

Conclusion

6.1. Summary and Conclusions

This thesis describes the verification of safety and liveness of the alternating bit protocol, a simple, but non-trivial protocol for the transfer of data from one party to another, in one direction only. The protocol is specified in an extended finite state machine protocol description language. This specification is translated, by hand, to first order predicate logic statements. These statements are written in the notation of the MIZAR-2 language.

Proofs of safety and liveness are recorded in the MIZAR-2 language. The entire set of MIZAR-2 texts is checked for accurate syntax and correctness of proofs by the MIZAR-2 processor.

During the verification of the alternating bit protocol an assertion which relied on the nature of the protocol was used to eliminate one of the liveness proofs. This assertion required that the liveness and safety procedures be separable. That is, that the proofs of safety and liveness be written separately and further, that the proof of safety for each system state could be proved separately.

The translation to MIZAR-2 is based on a known technique of recording programming language semantics. This technique had to be extended to support the concurrent operation of two parties and operate in an environment controlled by time. To this end the control point concept had to be extended to describe two simultaneous points of execution and it, as well as the variable histories, had to be referenced against time.

We find the proposed approach interesting and suggest the following extensions.

6.2. Further Areas of Study

The axiomatization of PDL in MIZAR-2, as shown in Appendix A3, is imperfect in that it was not completely general. That is, a new axiomatization may be required for a different protocol. Thus, one possible area of research is to create a general purpose axiomatization of PDL in MIZAR-2. This axiomatization would describe the MIZAR-2 statements equivalent to all PDL statements. This axiomatization could be used to create a MIZAR-2 description of any PDL specification in a mechanical way.

A related area is to create a Describer [Rud87] program which will create a MIZAR-2 description of a PDL specification. The describer program would read the PDL specification and output the axiomatic MIZAR-2 description of the specification, as shown in Appendix A3. This would eliminate the step of hand-creating the axioms.

Another direction is to generate MIZAR-2 descriptions from Estelle specifications instead of PDL specifications. As described in section 1.5, PDL is similar to Estelle in many respects. Much research has been done on the verification and analysis of protocol specifications in Estelle. This includes the automatic generation of the system state reachability graph, which is required for the proof of safety, as described in section 4.4. For this reason, it could be very useful to work with Estelle specifications using MIZAR-2 in the same way that PDL specifications were examined in this paper.

References

- [B87] ISO Subgroup B.
Estelle - A Formal Description Technique Based on an Extended State Transition Model.
ISO, Paris, March, 1987.
- [BIS86] Thomas P. Blumer and Deepinder P. Sidhu.
Mechanical Verification and Automatic Implementation of Communication Protocols.
IEEE Transactions on Software Engineering, August, 1986.
- [Boc86] Gregor V. Bochman.
Recent Developments in Protocol Specification, Validation and Testing.
Universite de Montreal, January, 1986.
- [Boc87] Gregor V. Bochman.
Specification of A Simplified Transport Protocol Using Different Formal Description Techniques.
Universite de Montreal, April, 1987.
- [BoS80] Gregor V. Bochmann and Carl A. Sunshine.
Formal Methods in Communication Protocol Design.
IEEE Transactions On Communications, April, 1980.
- [CCI] CCITT.
SDL Description.
Z 101, CCITT.
- [CaR82] Claudio Carrelli and D. J. Roche.
CCITT Languages for SPC Switching Systems.
IEEE Transactions on Communications, June, 1982.
- [HaO83] Brent T. Hailpern and Susan S. Owicki.
Modular Verification of Computer Communication Protocols.
IEEE Transactions on Communications, January, 1983.
- [JeW78] K. Jensen and N. Wirth.
PASCAL User Manual and Report, 2nd edition.
Springer-Verlag, New York, 1978.
- [KeR78] B. W. Kernighan and D. M. Ritchie.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- [Mer79] Philip M. Merlin.
Specification and Validation of Protocols.
IEEE Transaction on Communications, November, 1979.
- [MLP79] Richard A. De Millo, Richard J. Lipton and Alan J. Perlis.
Social Processes and Proofs of Theorems and Programs.
Communications of the Association for Computing Machinery, 22(5), May 1979.
- [Pet81] J.L. Peterson.
Petri Net Theory and the Modeling of Systems.
Prentice-Hall, New York, 1981.

- [Pet62] C. A. Petri.
Kommunikation mit Automaten.
Schriften des Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik
an der Universität Bonn, Bonn, West Germany, 1962.
- [PrR83] K. Prazmowski and P. Rudnicki.
MIZAR-MSE PRIMER, A Draft.
Institute of Computer Science, Polish Academy of Science, October, 1983.
- [PrR88] K. Prazmowski and P. Rudnicki.
MIZAR-MSE Primer and User Guide.
University of Alberta, July 1988.
- [Rud83] Harry Rudin.
An Informal Overview of Formal Protocol Specification.
IEEE Communications, March, 1983.
- [RuD85] Piotr Rudnicki and Włodzimierz Drabent.
Proving Properties of Pascal Programs in Mizar 2.
Acta Informatica, 1985.
- [Rud87] Piotr Rudnicki.
What Should be Proved and Tested Symbolically in Formal Specifications.
Fourth International Workshop on Software Specification and Design, April 3-4, 1987.
- [Saj] Michał Sajkowski.
Protocol Verification Techniques: Status Quo and Perspectives.
pp. 697-720 in *Protocol Specification and Verification*, ed. Y. Yemini, R. Strom
and S. Yemini.
North-Holland, New York.
- [Sha86a] A. Udaya Shankar.
Verified Data Transfer Protocols With Variable Flow Control.
University of Maryland, December, 1986.
- [Sha86b] A. Udaya Shankar.
Time-Dependent Distributed Systems: Proving Safety, Liveness and Real-Time Properties.
University of Maryland, October, 1986.
- [Sun79] Carl A. Sunshine.
Formal Techniques for Protocol Specification and Verification.
IEEE Computer, September, 1979.
- [TsV88] George K. Tsiknis and Son T. Vuong.
Protocol Specification and Verification using The Significant Event Temporal Logic.
CIPS 1988, Edmonton, Alberta, 1988.
- [VHC86] Son T. Vuong, Daniel D. Hui and Don D. Cowan.
VALIRA - A Tool for Protocol Validation Via Reachability Analysis.
Proceedings of the Fifth IFIP Workshop on Protocol Specification, Verification
and Testing, Gray Rock, Montreal, 1986.

- [ZWR80] Pitro Zafiropulo, Colin H. West, Harry Rudin, D. D. Cowan and Daniel Brand.
Towards Analyzing and Synthesizing Protocols.
IEEE Transactions on Communications, April, 1980.

Appendix A1

Extensions To MIZAR-2

To facilitate verification of PDL specifications several extensions were made to the MIZAR-2 processor. The extensions fall into two groups, those which were to remedy some limitations of the implementation, which we shall call lexical modifications, and those which extend the notational convenience of MIZAR-2.

The lexical extensions were implemented as modifications and extensions to the existing MIZAR-2 source code.

The notational extensions were implemented as pre-compilation processors and utilized existing UNIX utilities and custom written Pascal [JeW78] code.

The extensions were made to the BSD UNIX 4.3 version, originally implemented in Pascal P8000 on the IBM 370 by Czesław Bylinski - leading implementer, Henryk Oryszczyszyn, Piotr Rudnicki and Andrzej Trybulec - head of the project, ported to BSD UNIX 4.3 by Piotr Rudnicki.

1.1. Lexical Modifications

The syntactic modifications allowed tabs, identifiers up to 32 characters, the underscore character in identifiers and lines up to 511 characters to be used. The error message facility has also been enhanced to provide textual error messages, in most cases, as opposed to just numbers.

1.2. Notational Modifications

Two notational modifications were made to make proofs of PDL specifications substantially shorter.

1.2.1. Function Application Notation

The first was to support an application notation. The application notation allows a function to be applied a constant number of times, without re-writing the name of the function. For example, if a specification required the application of the function "Nx" to the result of "Nx" applied to a variable T one could write either $Nx(Nx(T))$ or $Nx\{2\}(T)$.

Note that the application notation is only meaningful when a function has only one argument and its argument and result are of the same type.

The function application notation was implemented by a pre-compilation phase which translated the application notation into standard MIZAR-2. Thus, $Nx\{2\}(T)$ is translated into $Nx(Nx(T))$.

The pre-compiler is written in Pascal [JeW78].

1.2.2. Alternative Form of Predicate Definitions

In MIZAR-2 it is sometimes convenient and more readable to define one predicate in terms of another. Figure A1.1 shows a fragment of a MIZAR-2 text where the predicate denoting "a person is ready to go outdoors in a Northern Canadian Winter" is defined in terms of other predicates.

Note that the definitions of the predicates `Hat_On`, `Boots_On`, `Mitts_On` and `Coat_On` are not shown.

The definition of the predicate `Ready_To_Go` can be considered easy to understand if we understand the predicates which make up `Ready_To_Go`. If the `Hat_On`, `Boots_On`,

```
for P being Person pred Ready_To_Go;  
for P being Person pred Hat_On;  
for P being Person pred Boots_On;  
for P being Person pred Mitts_On;  
for P being Person pred Coat_On;  
  
Ready_To_Go_Definition:  
  for P being Person st Ready_To_Go[P]  
    holds  
      Hat_On[P] &  
      Boots_On[P] &  
      Mitts_On[P] &  
      Coat_On[P];
```

Figure A1.1 Predicate Ready_To_Go in Standard MIZAR-2

Mitts_On and Coat_On predicates are used elsewhere where it makes the specification easier to maintain as well. In this sense, we have modularized our specification.

The difficulty arises when this definition is used in a proof. As one would expect, to prove that a person is Ready_To_Go, one must first prove that all of the predicates referenced by Ready_To_Go_Definition are true.

Because of certain constraints of the MIZAR-2 processor, this must be done as four lemmas, inside a larger proof for Ready_To_Go which simply refers to the lemmas. If each of the predicates are also defined in terms of other predicates, each lemma must have its own lemmas. Very soon, in our experience, the advantages of having modularized the specification are lost in the extra effort required during the proofs which reference the definitions.

We have eliminated the effort required during the proof stage by utilizing the UNIX C language processor [KeR78] before the MIZAR-2 processor. Predicates which are used by

other predicates are defined to the pre-processor, which then substitutes the text of the predicate in every place that it is used. Figure A1.2 shows our example using the pre-processor facility.

```

#define Hat_On(P) ...
#define Boots_On(P) ...
#define Mitts_On(P) ...
#define Coat_On(P) ...

for P being Person pred Ready_To_Go;

Ready_To_Go_Definition:
  for P being Person st Ready_To_Go[P]
    holds
      Hat_On(P) &
      Boots_On(P) &
      Mitts_On(P) &
      Coat_On(P);

```

Figure A1.2 Predicate Ready_To_Go Using Pre-Processor

The text seen by the MIZAR-2 processor has no reference to `Hat_On`, etc. Instead the body of their definitions have been substituted. Thus, a proof may reference the definitions of `Hat_On`, etc as though they were actually direct attribute of the `Ready_To_Go` predicate, eliminating many proof steps.

Note that one a deficiency of this method is that the predicates `Hat_On`, `Boots_On`, etc, no longer use "[" and "]" to enclose their parameters. Unfortunately, this is how functions are specified in standard MIZAR-2. Note that their context allows us to unambiguously determine that they are predicates and not functions.

Appendix A2

Example PDL Specification

```

!
! Alternating bit protocol.
!
! This version is from Hailpern and Owicki,
! "Modular Verification of Computer Communication Protocols"
! IEEE Transactions on Communications,
! Volume COM-31, Number 1, January, 1983.
!
! This text is recorded in the Protocol Description Language
! described in Chapter 3.
!
protocol ALTERNATING_BIT
  configuration
    channel MAB  A transmit;
                  B receive;
    channel MBA  B transmit;
                  A receive;
  end_configuration

  party A
    in_client SOURCE;
    local DATA_TRANSMITTED;
    party_init
      read_client(SOURCE, DATA_TRANSMITTED);

      next_state TRANSMIT_ODD;
    end_party_init

    state TRANSMIT_ODD
      state_init
        transmit "1" || DATA_TRANSMITTED;
      end_state_init

      wait_event;

      receive
        "ACK1";

        read_client(SOURCE, DATA_TRANSMITTED);
        next_state TRANSMIT_EVEN;
      end_receive

!
! When other message received, retransmit the
! data and bit.
!

```

```

        unspecified
            next_state TRANSMIT_ODD;

    end_state TRANSMIT_ODD

state TRANSMIT_EVEN
    state_init
        transmit "0" || DATA_TRANSMITTED;
    end_state_init

    wait_event;

    receive
        "ACK0";

        read_client(SOURCE, DATA_TRANSMITTED);
        next_state TRANSMIT_ODD;
    end_receive

!
! When other message received, retransmit the
! data and bit.
!
    unspecified
        next_state TRANSMIT_EVEN;
    end_state TRANSMIT_EVEN
end_party A

party B
    out_client SINK;
    local DATA_RECEIVED;
    party_init
        next_state RECEIVE_ODD;
    end_party_init

state RECEIVE_ODD
    state_init

    end_state_init

    wait_event;

    receive
        "1" || DATA_RECEIVED;

        write_client(SINK, DATA_RECEIVED);
        transmit "ACK1";
        next_state RECEIVE_EVEN;
    end_receive

```

```

!
! When other message received, retransmit the
! acknowledgement.
!
    unspecified
        transmit "ACK0";
        next_state RECEIVE_ODD

end_state RECEIVE_ODD

state RECEIVE_EVEN
    state_init

    end_state_init

    wait_event;

    receive
        "0" || DATA_RECEIVED;

        write_client(SINK, DATA_RECEIVED);
        transmit "ACK0";
        next_state RECEIVE_ODD;
    end_receive

!
! When other message received, retransmit the
! acknowledgement.
!
    unspecified
        transmit "ACK1";
        next_state same

    end_state RECEIVE_EVEN
end_party B

end_protocol ALTERNATING_BIT

```

Appendix A3 Alternating Bit Protocol in MIZAR-2

3.1. Introduction

This appendix contains a specification, in MIZAR-2, of the alternating bit protocol

3.2. Symbols Used In This Specification

The following mechanisms are used in the specification:

3.2.1. Time

Time is considered as discrete time instants. Time instants are modeled as natural numbers. The passage of time is denoted by the function Nx . The time instant following the current time instant (T) is denoted by $Nx(T)$. In general $Nx\{N\}(T) = T + N$.

3.2.2. Local Variables

Local variables are described using histories. A history is a function which returns the value of the variable at any instant in time. For each local variable in the specification, a function, whose only parameter is time, is declared. The function has the same name as the local variable.

For example, the local variable `DATA_TO_BE_TRANSMITTED` is modeled by the function declaration:

for T being natural

consider `DATA_TO_BE_TRANSMITTED` being String;

The value of `DATA_TO_BE_TRANSMITTED` at time T is given by the expression:

$\text{DATA_TO_BE_TRANSMITTED}(T)$

The value of $\text{DATA_TO_BE_TRANSMITTED}$ in the next time instant is given by the expression:

$\text{DATA_TO_BE_TRANSMITTED}(N_x(T))$

All variables are described as strings.

3.2.3. Protocol Messages

Protocol messages are modeled by a pair (String, Bit) where String is the message to be transmitted and Bit is the control bit of the transmission. String may take on any string value while Bit may take on the values ACK0, ACK1 and CorruptedBit in transmissions from B to A and the values ZeroString, OneString and CorruptedBit in transmissions from A to B.

Transmissions in either direction are considered successful if the bit is not set to CorruptedBit upon reception.

3.2.4. Control Points

Control points are modeled as histories with respect to time. There is a separate control point for each party. The function ControlPoint maps the party and time to a control point.

3.2.5. Channels

Channels are modeled as histories which map time to a channel. Since, in the alternating bit protocol, the channel can only contain one message, the channel is either full or empty at each time instant. If the channel is full the contents of the channel is a message.

There is a separate function defined for each channel. The parameter to this function is time. The function's value is a message, as described previously.

3.3. Outline

The first portion in this text are the definitions of all the types used in the proofs.

The first axiom definition is of the axioms for string matching. This simply state that a string matches any string composed of it and any other string.

The definition of the uniqueness of strings within a channel follows. The next two sections describe definitions used in the axiomatization of parties A and B. Following the definition section is the actual axiomatization for A and B.

Each control point of the PDL specification is specified by a separate axiom. Each axiom is numbered by the party followed by the control point.

environ

==

== Definitions used later in the specification.

==

let i, j, N denote natural;

let T, T' denote natural; == (type time)

for i, j reconsider $i + j$ as natural;

for i, j being natural pred $i < j$;

let $CP, NextCP$ denote natural; == (ControlPointType)

type Channel; let C denote Channel;

```

type String; let S,S1,S2 denote String;

let M,MPRIME denote String;

let D,Bit denote String;

type Party; let P denote Party;

for P,T consider ControlPoint being natural;

==
== The definition of string concatenation.
== This function is required because the parity bit
== and the message are concatenated during transmission.
==
    for S1,S2 consider concat being String;
==
== Definition of a match between a pattern and
== a received message.
== A message matches a pattern if either of the
== concatenated strings which constitute the message
== match the target pattern.
== The axiom AxMatch indicates that the string S,
== which consists of the concatenation of S1 and S2,
== matches both S1 and S2.
==
== The axiom AxMatchExact is used when the string

    for M,S pred match;

    AxMatch:for S,S1,S2 st S = concat(S1,S2)
        holds match[S, S1] & match[S,S2];

    AxMatchExact:for S,S1 st S = S1
        holds match[S, S1] & match[S1, S];

==
== Definition of Client interaction Queues
==
== This is the queues which provide the source
== of information for the client.
==
== in_client SOURCE
==
    for i consider SOURCE being String;

    for T consider SOURCE_length being natural;

==
== This function is used to determine the contents

```



```

== of a channel.
==
    for C consider Contents being String;
==
== Channel definitions
==
== These are the definitions of the channels
== used to communicate between the two parties.
==
    for T consider MBA being Channel;

    for T consider MAB being Channel;
==
== Definition of local variables
==
== local DATA_TO_BE_TRANSMITTED
==

    for T consider DATA_TO_BE_TRANSMITTED being String;

==
== Definition of the output queue.
==
== Notice that unlike the input queue, the output
== queue varies with respect to time.
==
== out_client SINK
==

    for T,i consider SINK being String;

    for T consider SINK_length being natural;

==
== local DATA_RECEIVED
==
    for T consider DATA_RECEIVED being String;

==
== Definition of semantics of channels.
==
== Full and Empty determine whether a specific
== channel is Full or Empty.
==
    for C pred Full;

    for C pred Empty;

==
== A channel cannot be both full and empty at the

```

```

== same time.
==
    FullOrEmpty: for C holds (Full[C] or Empty[C]) &
                    not (Full[C] & Empty[C]);

==
== The Message function composes a messages from its
== parameters, the data string and the control bit.
==
    for D, Bit consider Message being String;

==
== The Data function extracts the data string from a message.
==
    for S consider Data being String;

==
== The ControlBit function extracts the control bit a message.
==
    for S consider ControlBit being String;

==
== The axiom MessageDefinition describes the relationship
== between a message and its control bit and data string.
==

    MessageDefinition: for D, Bit, S
        st
            Message(D, Bit) = S
        holds
            Data(S) = D &
            ControlBit(S) = Bit;

==
== Once SINK has taken been assigned a value,
== that value does not change.
==
    SinkUnchanged: for T, T', N
        holds
            SINK(T, N) = SINK(T', N);

==
== Definition of next time instant.
==

    for T consider Nx being natural;

==
== Definition of Parties in this protocol definition.
==
    given A, B being Party;

```

```

==
== Definition of unique strings used in this specification.
==

```

```

    given ACK0,ACK1,ZeroString,OneString,
           EmptyString,CorruptedBit
           being String such that
           StringInequality:ZeroString <> CorruptedBit &
                               OneString <> CorruptedBit &
                               ACK1 <> CorruptedBit &
                               ACK0 <> CorruptedBit &
                               ACK1 <> ACK0 &
                               OneString <> ZeroString;

```

```

==
== Semantics of PDL statements.
==
== Definition of predicates
==

```

== Party A ==

```

==
== Variable axioms
==
#define SOURCE_length_unchanged(T)
    SOURCE_length(T) = SOURCE_length(Nx(T))

#define ALocalsUnchanged(T)
    (DATA_TO_BE_TRANSMITTED(T)
     = DATA_TO_BE_TRANSMITTED(Nx(T)) &
     SOURCE_length_unchanged(T))

==
== Channel axioms
==
#define MBA_For_A_Unchanged(T)
    (Full[MBA(T)] implies (Full[MBA(Nx(T))] &
    (Contents(MBA(T)) = Contents(MBA(Nx(T))))))

#define MAB_For_A_Unchanged(T)
    (Empty[MAB(T)] implies Empty[MAB(Nx(T))])

#define AChannelsUnchanged(T)
    (MBA_For_A_Unchanged(T) & MAB_For_A_Unchanged(T))

==
== Program Statement Axioms
==

```

```

==
== Only the current message changes in the source queue.
== The queue itself remains the same.
==
#define Progress_Read(T)
    DATA_TO_BE_TRANSMITTED(Nx(T))
    = SOURCE(SOURCE_length(Nx(T))) &
    SOURCE_length(T) + 1 = SOURCE_length(Nx(T))

#define Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED(T,NextCP)
    Progress_Read(T) &
    ControlPoint(A, Nx(T)) = NextCP

#ifdef PERFECT_CHANNELS

#define ATransmit(T,D,Bit,NextCP) (Full[MAB(Nx(T))] &
    Contents(MAB(Nx(T))) = Message(D,Bit) &
    ControlPoint(A, Nx(T)) = NextCP &
    MBA_For_A_Unchanged(T) & ALocalsUnchanged(T))

#else

#define ATransmit(T,D,Bit,NextCP) (Full[MAB(Nx(T))] &
    Contents(MAB(Nx(T))) = Message(D,Bit) &
    (Contents(MAB(Nx(T))) = Message(D,Bit) or
    Contents(MAB(Nx(T))) = Message(D,CorruptedBit)) &
    ControlPoint(A, Nx(T)) = NextCP &
    MBA_For_A_Unchanged(T) & ALocalsUnchanged(T))

#endif

== Party B ==

==
== Variable axioms
==
#define SINK_length_unchanged(T)
    (SINK_length(T) = SINK_length(Nx(T)))

#define BLocalsUnchanged(T)
    (DATA_RECEIVED(T) = DATA_RECEIVED(Nx(T)) &
    SINK_length_unchanged(T))

==
== Channel axioms
==
#define MAB_For_B_Unchanged(T)
    (Full[MAB(T)] implies ( Full[MAB(Nx(T))] &
    (Contents(MAB(T)) = Contents(MAB(Nx(T))))))

```

```

#define MBA_For_B_Unchanged(T)
  (Empty[MBA(T)] implies Empty[MBA(Nx(T))])

#define BChannelsUnchanged(T)
  (MBA_For_B_Unchanged(T) & MAB_For_B_Unchanged(T))

==
== Program Statement Axioms
==

#define Progress_Write(T)
  Data(Contents(MBA(T))) = SINK(Nx(T), SINK_length(T)) &
  (SINK_length(Nx(T)) = SINK_length(T) + 1)

#define Write_Client_SINK_DATA_RECEIVED(T, NextCP)
  Progress_Write(T) & ControlPoint(B, Nx(T))=NextCP

#ifdef PERFECT_CHANNELS

#define BTransmit(T,D,Bit,NextCP) (Full[MBA(Nx(T))] &
  Contents(MBA(Nx(T))) = Message(D,Bit) &
  ControlPoint(B, Nx(T)) = NextCP &
  MAB_For_B_Unchanged(T) & BLocalsUnchanged(T))

#else

#define BTransmit(T,D,Bit,NextCP) (Full[MBA(Nx(T))] &
  (Contents(MBA(Nx(T))) = Message(D,Bit) or
  Contents(MBA(Nx(T))) = Message(D, CorruptedBit)) &
  ControlPoint(B, Nx(T)) = NextCP &
  MAB_For_B_Unchanged(T) & BLocalsUnchanged(T))

#endif

===== Program Specification =====
A1:for T st ControlPoint(A, T) = 1
  holds Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED(T, 2)
  & AChannelsUnchanged(T);

A2:for T st ControlPoint(A, T) = 2
  holds ATransmit(T, DATA_TO_BE_TRANSMITTED(T), OneString, 3);

A3x1:for T st ControlPoint(A, T)= 3 &
  ControlBit(Contents(MBA(T))) = ACK1
  holds ControlPoint(A, Nx(T)) = 4 &
  Empty[MBA(Nx(T))] & MAB_For_A_Unchanged(T) &
  Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED(T, 4);

A3x2:for T st ControlPoint(A, T)=3
  & ControlBit(Contents(MBA(T))) <> ACK1

```

```

    holds ControlPoint(A, Nx(T)) = 2 &
    Empty[MBA(Nx(T))] & ALocalsUnchanged(T) &
    MAB_For_A_Unchanged(T);

A3x3:for T st ControlPoint(A, T)= 3 & Empty[MBA(T)]
    holds ControlPoint(A, Nx(T)) = 3 &
    ALocalsUnchanged(T) & AChannelsUnchanged(T);

A4:for T st ControlPoint(A, T) = 4
    holds ATransmit(T, DATA_TO_BE_TRANSMITTED(T), ZeroString, 5);

A5x1:for T st ControlPoint(A, T)=5 &
    ControlBit(Contents(MBA(T))) = ACK0
    holds ControlPoint(A, Nx(T)) = 5 &
    Empty[MBA(Nx(T))] & MAB_For_A_Unchanged(T) &
    Read_Client_SOURCE_DATA_TO_BE_TRANSMITTED(T, 2);

A5x2:for T st ControlPoint(A, T) = 5 &
    ControlBit(Contents(MBA(T))) <> ACK0
    holds ControlPoint(A, Nx(T)) = 4 &
    Empty[MBA(Nx(T))] & ALocalsUnchanged(T) &
    MAB_For_A_Unchanged(T);

A5x3:for T st ControlPoint(A, T)= 5 & Empty[MBA(T)]
    holds ControlPoint(A, Nx(T)) = 5 &
    ALocalsUnchanged(T) & AChannelsUnchanged(T);

==
== B Party
==

B1x1:for T st ControlPoint(B, T) = 1 &
    ControlBit(Contents(MAB(T))) = OneString
    holds Empty[MAB(Nx(T))] &
    DATA_RECEIVED(Nx(T)) = Data(Contents(MAB(T))) &
    MBA_For_B_Unchanged(T) &
    Write_Client_SINK_DATA_RECEIVED(T, 2);

B1x2:for T st ControlPoint(B, T)=1 &
    ControlBit(Contents(MAB(T))) <> OneString
    holds Empty[MAB(Nx(T))] &
    ControlPoint(B, Nx(T)) = 3 &
    SINK_length_unchanged(T) &
    MBA_For_B_Unchanged(T);

B1x3:for T st ControlPoint(B, T) = 1 & Empty[MAB(T)]
    holds ControlPoint(B, Nx(T))= ControlPoint(B, T) &
    SINK_length_unchanged(T) &
    BChannelsUnchanged(T);

```

```

B2:for T st ControlPoint(B, T) = 2
    holds BTransmit(T, EmptyString, ACK1, 4);

B3:for T st ControlPoint(B, T) = 3
    holds BTransmit(T, EmptyString, ACK0, 1);

B4x1:for T st ControlPoint(B, T)=4 &
    ControlBit(Contents(MAB(T))) = ZeroString
    holds
        Empty[MAB(Nx(T))] &
        DATA_RECEIVED(Nx(T)) = Data(Contents(MAB(T))) &
        MBA_For_B_Unchanged(T) &
        Write_Client_SINK_DATA_RECEIVED(T, 5);

B4x2: for T st ControlPoint(B, T)=4 &
    ControlBit(Contents(MAB(T))) <> ZeroString
    holds
        Empty[MAB(Nx(T))] &
        ControlPoint(B, Nx(T)) = 6 &
        SINK_length_unchanged(T) &
        MBA_For_B_Unchanged(T);

B4x3: for T st ControlPoint(B, T)=4 & Empty[MAB(T)]
    holds
        ControlPoint(B, Nx(T)) = 4 &
        SINK_length_unchanged(T) &
        BChannelsUnchanged(T);

B5: for T st ControlPoint(B, T) = 5
    holds BTransmit(T, EmptyString, ACK0, 1);

B6:for T st ControlPoint(B, T) = 6
    holds BTransmit(T, EmptyString, ACK1, 4);

```

Appendix A4

Safety Proof for Alternating Bit Protocol

4.1. Overview

The proof of safety has eight lemmas, one for each of the possible state transitions in the protocol.

4.2. Proof Structure

Before each proof there is a comment block indicating the transition described. The comment block shows the line number that each party is in before and after the transition, as well as the contents of each of the channels. The status before the transmission is in the first column, while the status after the transmission is shown in the second column.

If the transition is part of the liveness procedure then the name of the proof is TransitionX where X is 1 to 4.

If the transition is one of the safety procedures then the name of the proof is TransitionXCorrupt where X is 1 to 4.

TransitionXCorrupt describes the safety procedure when the transmission for TransitionX fails.

Note that safety transitions will always begin with a corrupt message in the channel and complete with the uncorrupted message in the channel.

```
#include "altbit.h"
begin
==
== A:3<One>   A:3
== B:1        B:4<Ack 1>
```



```

==
Transition1:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 3 &
    Contents(MAB(T)) = Message(SOURCE(N), OneString) &
    SOURCE_length(T) = N &
    SINK_length(T) = M &
    ControlPoint(B, T) = 1 &
    Empty[MBA(T)];

    ControlBit(Contents(MAB(T))) = OneString
      by Assumptions, MessageDefinition;
  then S1:ControlPoint(B, Nx(T)) = 2 &
    Progress_Write(T) &
    Empty[MBA(Nx(T))] &
    Empty[MAB(Nx(T))] by B1x1, Assumptions;
  then Part1:ControlPoint(B, Nx{2}(T)) = 4 &
    (Contents(MBA(Nx{2}(T)))
      = Message(EmptyString, ACK1) or
      Contents(MBA(Nx{2}(T)))
      = Message(EmptyString, CorruptedBit)) &
    SINK_length(Nx{2}(T)) = M + 1 by B2, Assumptions;

    ControlPoint(A, Nx(T)) = 3 & SOURCE_length(Nx(T)) = N
      by Assumptions, A3x3;
  then ControlPoint(A, Nx{2}(T)) = 3 &
    SOURCE_length(Nx{2}(T)) = N by A3x3, S1;

  hence ex T' st
    ControlPoint(B, T') = 4 &
    (Contents(MBA(T')) = Message(EmptyString, ACK1) or
    Contents(MBA(T'))
      = Message(EmptyString, CorruptedBit)) &
    SINK_length(T') = M + 1 &
    SOURCE_length(T') = N &
    ControlPoint(A, T') = 3 by Part1;
  end;
==
== A:3      A:5<Zero>
== B:4<ACK1> B:4
==
Transition2:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 3 &
    Contents(MBA(T)) = Message(EmptyString, ACK1) &
    SOURCE_length(T) = N &

```

```

    SINK_length(T) = M &
    ControlPoint(B, T) = 4 &
    Empty[MAB(T)];

    ControlBit(Contents(MBA(T))) = ACK1
    by Assumptions, MessageDefinition;
    then S1:ControlPoint(A, Nx(T)) = 4 &
    Progress_Read(T) &
    Empty[MBA(Nx(T))] & Empty[MAB(Nx(T))]
    by A3x1, Assumptions;
    then S1':ControlPoint(A, Nx{2}(T)) = 5 &
    (Contents(MAB(Nx{2}(T))) =
    Message(DATA_TO_BE_TRANSMITTED(Nx{2}(T)),
    ZeroString) or
    Contents(MAB(Nx{2}(T))) =
    Message(DATA_TO_BE_TRANSMITTED(Nx{2}(T)),
    CorruptedBit)) &
    SOURCE_length(Nx{2}(T)) = N + 1
    by A4, Assumptions;
    then Part1:ControlPoint(A, Nx{2}(T)) = 5 &
    (Contents(MAB(Nx{2}(T)))
    = Message(SOURCE(N), ZeroString) or
    Contents(MAB(Nx{2}(T)))
    = Message(SOURCE(N), CorruptedBit)) &
    SOURCE_length(Nx{2}(T)) = N + 1
    by A4, Assumptions;

    ControlPoint(B, Nx(T)) = 4 & SINK_length(Nx(T)) = M
    by Assumptions, B4x3;
    then ControlPoint(B, Nx{2}(T)) = 4 &
    SINK_length(Nx{2}(T)) = M by B4x3, S1;

    hence ex T' st
    ControlPoint(A, T') = 5 &
    (Contents(MAB(T'))
    = Message(SOURCE(N), ZeroString) or
    Contents(MAB(T'))
    = Message(SOURCE(N), CorruptedBit)) &
    SINK_length(T') = M &
    SOURCE_length(T') = N + 1 &
    ControlPoint(B, T') = 4 by Part1;
  end;
==
== A:3<Corrupt>A:3<One>
== B:1      B:1
==
Transition1Corrupt:
now
  let T,N,M be natural such that
  Assumptions:

```

```

ControlPoint(A, T) = 3 &
Contents(MAB(T)) = Message(SOURCE(N), CorruptedBit) &
SOURCE_length(T) = N &
SINK_length(T) = M &
ControlPoint(B, T) = 1 &
Empty[MBA(T)];

ControlBit(Contents(MAB(T))) <> OneString
  by Assumptions, StringInequality, MessageDefinition;
then S1:ControlPoint(B, Nx(T)) = 3 &
  SINK_length(Nx(T)) = M &
  Empty[MBA(Nx(T))] &
  Empty[MAB(Nx(T))] by B1x2, Assumptions;
then S1':ControlPoint(B, Nx{2}(T)) = 1 &
  (Contents(MBA(Nx{2}(T)))
    = Message(EmptyString, ACK0) or
    Contents(MBA(Nx{2}(T)))
    = Message(EmptyString, CorruptedBit)) &
  SINK_length(Nx{2}(T)) = M by B3;
then ControlBit(Contents(MBA(Nx{2}(T)))) <> ACK1
  by StringInequality, MessageDefinition;
then Part1:ControlBit(Contents(MBA(Nx{2}(T)))) <> ACK1 &
  ControlPoint(B, Nx{2}(T)) = 1 &
  SINK_length(Nx{2}(T)) = M by S1';

S2:ControlPoint(A, Nx(T)) = 3 &
  SOURCE_length(Nx(T)) = N by Assumptions, A3x3;
then S3:ControlPoint(A, Nx{2}(T)) = 3 &
  Empty[MAB(Nx{2}(T))] & SOURCE_length(Nx{2}(T)) = N
  by A3x3, S1;
then S4:ControlPoint(A, Nx{3}(T)) = 2 &
  SOURCE_length(Nx{3}(T)) = N &
  Empty[MBA(Nx{3}(T))] &
  Empty[MAB(Nx{3}(T))] by A3x2, Part1;
then Part2:ControlPoint(A, Nx{4}(T)) = 3 &
  (Contents(MAB(Nx{4}(T))) =
  Message(DATA_TO_BE_TRANSMITTED(Nx{4}(T)), OneString)
  or Contents(MAB(Nx{4}(T))) =
  Message(DATA_TO_BE_TRANSMITTED(Nx{4}(T)),
    CorruptedBit)) &
  SOURCE_length(Nx{4}(T)) = N by A2;

ControlPoint(B, Nx{3}(T)) = 1 & SINK_length(Nx{3}(T)) = M
  by Part1, B1x3, S3;
then ControlPoint(B, Nx{4}(T)) = 1 &
  Empty[MBA(Nx{4}(T))] &
  SINK_length(Nx{4}(T)) = M by B1x3, S4;

hence ex T st
  ControlPoint(B, T) = 1 &

```

```

    Empty[MBA(T)] &
    SINK_length(T) = M &
    (Contents(MAB(T))
      = Message(DATA_TO_BE_TRANSMITTED(T), OneString)
      or Contents(MAB(T))
      = Message(DATA_TO_BE_TRANSMITTED(T), CorruptedBit))
    & ControlPoint(A, T) = 3 &
    SOURCE_length(T) = N by Part2;
  end;
==
== A:3      A:3
== B:4<Corrupt>B:4<ACK1>
==
  Transition2Corrupt:
  now
    let T,N,M be natural such that
    Assumptions:
      ControlPoint(A, T) = 3 &
      Empty[MAB(T)] &
      ControlPoint(B, T) = 4 &
      Contents(MBA(T))
        = Message(EmptyString, CorruptedBit) &
      SOURCE_length(T) = N &
      SINK_length(T) = M;

    ControlBit(Contents(MBA(T))) <> ACK1
      by Assumptions, StringInequality, MessageDefinition;
    then S1:ControlPoint(A, Nx(T)) = 2 &
      SOURCE_length(Nx(T)) = N &
      Empty[MBA(Nx(T))] &
      Empty[MAB(Nx(T))] by A3x2, Assumptions;
    then S1':ControlPoint(A, Nx{2}(T)) = 3 &
      SOURCE_length(Nx{2}(T)) = N &
      (Contents(MAB(Nx{2}(T)))
        = Message(DATA_TO_BE_TRANSMITTED(Nx(T)), OneString) or
        Contents(MAB(Nx{2}(T)))
        = Message(DATA_TO_BE_TRANSMITTED(Nx(T)), CorruptedBit))
      by A2;
    then ControlBit(Contents(MAB(Nx{2}(T)))) <> ZeroString
      by StringInequality, MessageDefinition;
    then Part1:
      ControlBit(Contents(MAB(Nx{2}(T)))) <> ZeroString &
      SOURCE_length(Nx{2}(T)) = N &
      ControlPoint(A, Nx{2}(T)) = 3 by S1';

    S2:ControlPoint(B, Nx(T)) = 4 & SINK_length(Nx(T)) = M
      by Assumptions, B4x3;
    then S3:ControlPoint(B, Nx{2}(T)) = 4 &
      SINK_length(Nx{2}(T)) = M &
      Empty[MBA(Nx{2}(T))] by B4x3, S1;

```

```

then S4:ControlPoint(B, Nx{3}(T)) = 6 &
  SINK_length(Nx{3}(T)) = M & Empty[MBA(Nx{3}(T))] &
  Empty[MAB(Nx{3}(T))] by B4x2, Part1;

then Part2:ControlPoint(B, Nx{4}(T)) = 4 &
  SINK_length(Nx{4}(T)) = M &
  (Contents(MBA(Nx{4}(T))) =
    Message(EmptyString, ACK1) or
    Contents(MBA(Nx{4}(T))) =
      Message(EmptyString, CorruptedBit)) by B6;

ControlPoint(A, Nx{3}(T)) = 3 &
  SOURCE_length(Nx{3}(T)) = N
  by Part1, A3x3, S3;
then ControlPoint(A, Nx{4}(T)) = 3 &
  Empty[MAB(Nx{4}(T))] &
  SOURCE_length(Nx{4}(T)) = N by A3x3, S4;

hence ex T' st
  ControlPoint(A, T') = 3 &
  Empty[MAB(T')] &
  SOURCE_length(T') = N &
  ControlPoint(B, T') = 4 & SINK_length(T') = M &
  (Contents(MBA(T'))
    = Message(EmptyString, ACK1) or
    Contents(MBA(T'))
    = Message(EmptyString, CorruptedBit)) by Part2;
end;
==
== A:5<Zero> A:5
== B:4      B:1<ACK0>
==
Transition3:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 5 &
    Contents(MAB(T)) = Message(SOURCE(N), ZeroString) &
    SOURCE_length(T) = N &
    SINK_length(T) = M &
    ControlPoint(B, T) = 4 &
    Empty[MBA(T)];

  ControlBit(Contents(MAB(T))) = ZeroString
  by Assumptions, MessageDefinition;
then S1:ControlPoint(B, Nx(T)) = 5 &
  Progress_Write(T) &
  Empty[MBA(Nx(T))] &
  Empty[MAB(Nx(T))] by B4x1, Assumptions;
then Part1:ControlPoint(B, Nx{2}(T)) = 1 &

```

```

    (Contents(MBA(Nx{2}(T)))
      = Message(EmptyString, ACK0) or
      Contents(MBA(Nx{2}(T)))
      = Message(EmptyString, CorruptedBit)) &
    SINK_length(Nx{2}(T)) = M + 1 by B5, Assumptions;

  ControlPoint(A, Nx(T)) = 5
    & SOURCE_length(Nx(T)) = N by Assumptions, A5x3;
  then ControlPoint(A, Nx{2}(T)) = 5
    & SOURCE_length(Nx{2}(T)) = N by A5x3, S1;

  hence ex T' st
    ControlPoint(B, T') = 1 &
    (Contents(MBA(T'))
      = Message(EmptyString, ACK0) or
      Contents(MBA(T'))
      = Message(EmptyString, CorruptedBit)) &
    SINK_length(T') = M + 1 &
    SOURCE_length(T') = N &
    ControlPoint(A, T') = 5 by Part1;
end;

==
== A:5<Corrupt>A:5<Zero>
== B:4      B:4
==
Transition3Corrupt:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 5 &
    Contents(MAB(T)) = Message(SOURCE(N), CorruptedBit) &
    SOURCE_length(T) = N &
    SINK_length(T) = M &
    ControlPoint(B, T) = 4 &
    Empty[MBA(T)];

  ControlBit(Contents(MAB(T))) <> ZeroString
    by Assumptions, StringInequality, MessageDefinition;
  then S1:ControlPoint(B, Nx(T)) = 6 &
    SINK_length(Nx(T)) = M &
    Empty[MBA(Nx(T))] &
    Empty[MAB(Nx(T))] by B4x2, Assumptions;
  then S1':ControlPoint(B, Nx{2}(T)) = 4 &
    SINK_length(Nx{2}(T)) = M &
    (Contents(MBA(Nx{2}(T)))
      = Message(EmptyString, ACK1) or
      Contents(MBA(Nx{2}(T)))
      = Message(EmptyString, CorruptedBit)) by B6;
  then ControlBit(Contents(MBA(Nx{2}(T)))) <> ACK0

```

```

    by StringInequality, MessageDefinition;
  then Part1: ControlBit(Contents(MBA(Nx{2}(T)))) <> ACK0 &
    SINK_length(Nx{2}(T)) = M &
    ControlPoint(B, Nx{2}(T)) = 4 by S1';

  S2: ControlPoint(A, Nx(T)) = 5 &
    SOURCE_length(Nx(T)) = N by Assumptions, A5x3;
  then S3: ControlPoint(A, Nx{2}(T)) = 5 &
    SOURCE_length(Nx{2}(T)) = N &
    Empty[MAB(Nx{2}(T))] by A5x3, S1;

  then S4: ControlPoint(A, Nx{3}(T)) = 4 &
    SOURCE_length(Nx{3}(T)) = N &
    Empty[MBA(Nx{3}(T))] &
    Empty[MAB(Nx{3}(T))] by A5x2, Part1;

  then S4': ControlPoint(A, Nx{4}(T)) = 5 &
    SOURCE_length(Nx{4}(T)) = N &
    (Contents(MAB(Nx{4}(T))) =
      Message(DATA_TO_BE_TRANSMITTED(Nx{3}(T)), ZeroString) or
      Contents(MAB(Nx{4}(T))) =
      Message(DATA_TO_BE_TRANSMITTED(Nx{3}(T)), CorruptedBit))
    by A4;
  then Part2: ControlPoint(A, Nx{4}(T)) = 5 &
    SOURCE_length(Nx{4}(T)) = N &
    (Contents(MAB(Nx{4}(T)))
      = Message(SOURCE(N), ZeroString) or
      Contents(MAB(Nx{4}(T)))
      = Message(SOURCE(N), CorruptedBit)) by A4;

  ControlPoint(B, Nx{3}(T)) = 4 &
    SINK_length(Nx{3}(T)) = M by Part1, B4x3, S3;
  then ControlPoint(B, Nx{4}(T)) = 4 &
    SINK_length(Nx{4}(T)) = M &
    Empty[MBA(Nx{4}(T))] by B4x3, S4;

  hence ex T' st
    ControlPoint(B, T') = 4 &
    SINK_length(T') = M &
    Empty[MBA(T')] &
    (Contents(MAB(T'))
      = Message(SOURCE(N), ZeroString) or
      Contents(MAB(T'))
      = Message(SOURCE(N), CorruptedBit)) &
    SOURCE_length(T') = N &
    ControlPoint(A, T') = 5 by Part2;
end;

==
== A:5      A:3<One>

```

```

== B:1<ACK0> B:1
==
  Transition4:
  now
    let T,N,M be natural such that
    Assumptions:
      ControlPoint(A,T) = 5 &
      Contents(MBA(T)) = Message(EmptyString, ACK0) &
      SOURCE_length(T) = N &
      SINK_length(T) = M &
      ControlPoint(B, T) = 1 &
      Empty[MAB(T)];

    ControlBit(Contents(MBA(T))) = ACK0
      by Assumptions, MessageDefinition;
    then S1:ControlPoint(A, Nx(T)) = 2 &
      Progress_Read(T) &
      Empty[MBA(Nx(T))] &
      Empty[MAB(Nx(T))] by A5x1, Assumptions;
    then S1':ControlPoint(A, Nx{2}(T)) = 3 &
      SOURCE_length(Nx{2}(T)) = N + 1 &
      (Contents(MAB(Nx{2}(T))) =
        Message(DATA_TO_BE_TRANSMITTED(Nx(T)), OneString) or
        Contents(MAB(Nx{2}(T))) =
        Message(DATA_TO_BE_TRANSMITTED(Nx(T)), CorruptedBit))
      by A2;
    then Part1:ControlPoint(A, Nx{2}(T)) = 3 &
      SOURCE_length(Nx{2}(T)) = N + 1 &
      (Contents(MAB(Nx{2}(T)))
        = Message(SOURCE(N), OneString) or
        Contents(MAB(Nx{2}(T)))
        = Message(SOURCE(N), CorruptedBit))
      by A2;

    ControlPoint(B, Nx(T)) = 1 &
      SINK_length(Nx(T)) = M by Assumptions, B1x3;
    then ControlPoint(B, Nx{2}(T)) = 1 &
      SINK_length(Nx{2}(T)) = M by B1x3, S1;

    hence ex T' st
      ControlPoint(A, T') = 3 &
      SOURCE_length(T') = N + 1 &
      (Contents(MAB(T'))
        = Message(SOURCE(N), OneString) or
        Contents(MAB(T'))
        = Message(SOURCE(N), CorruptedBit)) &
      ControlPoint(B, T') = 1 &
      SINK_length(T') = M by Part1;
  end;
==

```



```

== A:5      A:5
== B:1<Corrupt>B:1<ACK0>
==
Transition4Corrupt:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 5 &
    Empty[MAB(T)] &
    ControlPoint(B, T) = 1 &
    Contents(MBA(T))
      = Message(EmptyString, CorruptedBit) &
    SOURCE_length(T) = N &
    SINK_length(T) = M;

  ControlBit(Contents(MBA(T))) <> ACK0
    by Assumptions, StringInequality, MessageDefinition;
  then S1:ControlPoint(A, Nx(T)) = 4 &
    SOURCE_length(Nx(T)) = N &
    Empty[MBA(Nx(T))] &
    Empty[MAB(Nx(T))] by A5x2, Assumptions;
  then S1':ControlPoint(A, Nx{2}(T)) = 5 &
    SOURCE_length(Nx{2}(T)) = N &
    (Contents(MAB(Nx{2}(T)))
      = Message(DATA_TO_BE_TRANSMITTED(Nx(T)),
        ZeroString) or
      Contents(MAB(Nx{2}(T)))
      = Message(DATA_TO_BE_TRANSMITTED(Nx(T)),
        CorruptedBit))
    by A4;
  then ControlBit(Contents(MAB(Nx{2}(T)))) <> OneString
    by StringInequality, MessageDefinition;
  then Part1:ControlBit(Contents(MAB(Nx{2}(T)))) <> OneString &
    SOURCE_length(Nx{2}(T)) = N &
    ControlPoint(A, Nx{2}(T)) = 5 by S1';

  S2:ControlPoint(B, Nx(T)) = 1 &
    SINK_length(Nx(T)) = M by Assumptions, B1x3;
  then S3:ControlPoint(B, Nx{2}(T)) = 1 &
    SINK_length(Nx{2}(T)) = M &
    Empty[MBA(Nx{2}(T))] by B1x3, S1;

  then S4:ControlPoint(B, Nx{3}(T)) = 3 &
    SINK_length(Nx{3}(T)) = M &
    Empty[MBA(Nx{3}(T))] &
    Empty[MAB(Nx{3}(T))] by B1x2, Part1;

  then Part2:ControlPoint(B, Nx{4}(T)) = 1 &
    SINK_length(Nx{4}(T)) = M &
    (Contents(MBA(Nx{4}(T))) =

```

```

    Message(EmptyString, ACK0) or
    Contents(MBA(Nx{4}(T))) =
        Message(EmptyString, CorruptedBit)) by B3;

ControlPoint(A, Nx{3}(T)) = 5 &
SOURCE_length(Nx{3}(T)) = N by Part1, A5x3, S3;
then ControlPoint(A, Nx{4}(T)) = 5 &
SOURCE_length(Nx{4}(T)) = N &
Empty[MAB(Nx{4}(T))] by A5x3, S4;

hence ex T' st
    ControlPoint(A, T') = 5 &
    Empty[MAB(T')] &
    SOURCE_length(T') = N &
    (Contents(MBA(T'))
        = Message(EmptyString, ACK0) or
        Contents(MBA(T'))
        = Message(EmptyString, CorruptedBit)) &
    SINK_length(T') = M &
    ControlPoint(B, T') = 1 by Part2;
end;
end

```

Appendix A5

Liveness Proof for Alternating Bit Protocol

5.1. Introduction

This appendix contains proof of liveness for the alternating bit protocol.

5.2. Proof Structure

The first interesting predicate described is the prefix predicate. Prefix describes the relationship between the input and output client queues. It states that to form a prefix of length N , the output queue must have the same first N messages as the input queue. These are denoted by SOURCE and SINK respectively. Note that the axioms PrefixDefinition1 and PrefixDefinition2 define the meaning of prefix while PrefixInduction provides a definition of prefix suited to an inductive proof.

Following these definitions are four lemmas, named Transition1, Transition2, Transition3 and Transition4. Each of these represents the one transition in the protocol system state transition graph.

Each lemma is structured in a very similar way. First the assumptions required are stated. These completely describe the state of the protocol before the transition. Following that is the body of the proof. The last part is the conclusion of the lemma, which completely describes the state of the protocol after the transition.

A notation is used to comment each transition. This notation indicates the control point of A and B along with the contents of the channels MAB and MBA respectively both before and after the transition. The state before the transition is in the first column, while the state of the protocol after the transition is noted in the second column.

For example, $A:5<ZERO>$ means that party A will be at control point 5 with a protocol message with the control bit set to zero in channel MAB. This is recorded in MIZAR-2 with the statements:

$ControlPoint(A,T) = 5$

and

$Contents(MAB(T)) = Message(SOURCE(N), ZeroString)$

After the four lemmas the proof of liveness is given. The liveness proof begins with the basis step:

ex T such that $prefix[1,T]$

The basis step begins by assuming that a prefix of 0 exists and then demonstrates that there is a time T at which a prefix of length 1 exists and the state of the protocol is exactly the assumptions required to demonstrate the proof of liveness.

The liveness proof then asserts the existence of four points in time, each of which correspond to one of the transitions. These are number T2 through T5. Following that, it is asserted that, based on the existence of T2 through T4, that at time T5 there is a prefix of length $N + 2$.

```
#define PERFECT_CHANNELS
#include "altbit.h"
```

```
==
== Definition of prefix used to show messages are transferred
== properly.
```

```
==
for N,I being natural pred prefix;
let T" denote natural;
```

```
PrefixInduction:for N,T,T",T"
```

```
st
```

```
    prefix[N,T] &
    SINK(T", N + 1) = SOURCE(N + 1) &
    SINK(T", ((N + 1) + 1)) = SOURCE((N + 1) + 1)
```

```
holds
```

```
    prefix[((N + 1) + 1), T"];
```

```

PrefixDefinition1:for N,T
  st
    prefix[N,T]
  holds
    SINK_length(T) = N &
    SOURCE_length(T) = N;

```

```

PrefixDefinition2:for N,T
  st
    SINK_length(T) = N &
    SOURCE_length(T) = N
  holds
    prefix[N,T];

```

```

PrefixForBasis:for N,T,T'
  st
    prefix[N,T] &
    SINK(T', N + 1) = SOURCE(N + 1)
  holds
    prefix[(N + 1), T'];

```

let T1,T2,T3,T4,T5 denote natural;

begin

Transition0:

now

let T,N,M be natural such that

Assumptions:

```

  prefix[0,T] &
  ControlPoint(A,T) = 1 &
  ControlPoint(B,T) = 1 &
  SOURCE_length(T) = N &
  SINK_length(T) = M &
  Empty[MBA(T)] &
  Empty[MAB(T)];

```

```

S1:ControlPoint(A, Nx(T)) = 2 &
  DATA_TO_BE_TRANSMITTED(Nx(T))
    = SOURCE(SOURCE_length(Nx(T))) &
  SOURCE_length(T) + 1 = SOURCE_length(Nx(T)) &
  Empty[MAB(Nx(T))] by A1, Assumptions;
then ControlPoint(A, Nx{2}(T)) = 3 &
  SOURCE_length(Nx{2}(T)) = N + 1 &
  Contents(MAB(Nx{2}(T))) =
    Message(DATA_TO_BE_TRANSMITTED(Nx(T)),
      OneString) by A2;
then Part1:ControlPoint(A, Nx{2}(T)) = 3 &
  SOURCE_length(Nx{2}(T)) = N + 1 &
  Contents(MAB(Nx{2}(T)))

```

```

    = Message(SOURCE(N + 1), OneString)
    by S1, Assumptions, A2;

```

```

ControlPoint(B, Nx(T)) = 1 &
  Empty[MBA(Nx(T))] &
  SINK_length(Nx(T)) = M by Assumptions, B1x3;
then ControlPoint(B, Nx{2}(T)) = 1 &
  SINK_length(Nx{2}(T)) = M &
  Empty[MBA(Nx{2}(T))] by B1x3, S1;

```

```

hence ex T' st
  ControlPoint(A, T') = 3 &
  SOURCE_length(T') = N + 1 &
  Contents(MAB(T')) =
    Message(SOURCE(N + 1), OneString) &
  ControlPoint(B, T') = 1 &
  SINK_length(T') = M &
  Empty[MBA(T')] by Part1;
end;

```

```

==
== A:3<One>   A:3
== B:1       B:4<Ack1>
==

```

Transition1:

now

let T,N,M be natural such that

Assumptions:

```

  ControlPoint(A, T) = 3 &
  Contents(MAB(T)) = Message(SOURCE(N), OneString) &
  SOURCE_length(T) = N &
  SINK_length(T) = M &
  ControlPoint(B, T) = 1 &
  Empty[MBA(T)];

```

```

ControlBit(Contents(MAB(T))) = OneString
  by Assumptions, MessageDefinition;
then S1:ControlPoint(B, Nx(T)) = 2 &
  SOURCE(N) = SINK(Nx(T), SINK_length(T)) &
  (SINK_length(Nx(T)) = SINK_length(T) + 1) &
  Empty[MBA(Nx(T))] &
  Empty[MBA(Nx(T))] by B1x1, Assumptions;
then Part1:ControlPoint(B, Nx{2}(T)) = 4 &
  SOURCE(N) = SINK(Nx{2}(T), SINK_length(Nx{2}(T))) &
  Contents(MBA(Nx{2}(T))) = Message(EmptyString, ACK1) &
  SINK_length(Nx{2}(T)) = M + 1 by B2, Assumptions;
then Part2:ControlPoint(B, Nx{2}(T)) = 4 &
  SOURCE(N) = SINK(Nx{2}(T), M + 1) &
  Contents(MBA(Nx{2}(T))) = Message(EmptyString, ACK1) &
  SINK_length(Nx{2}(T)) = M + 1 by B2, Assumptions;

```

```

ControlPoint(A, Nx(T)) = 3 & SOURCE_length(Nx(T)) = N
  by Assumptions, A3x3;
then ControlPoint(A, Nx{2}(T)) = 3 &
  SOURCE_length(Nx{2}(T)) = N &
  Empty[MAB(Nx{2}(T))]
  by A3x3, S1;

```

hence ex T' st

```

ControlPoint(B, T') = 4 &
SOURCE(N) = SINK(T', M + 1) &
Contents(MBA(T')) = Message(EmptyString, ACK1) &
SINK_length(T') = M + 1 &
SOURCE_length(T') = N &
ControlPoint(A, T') = 3 &
Empty[MAB(T')]
  by Part2;

```

end;

==

== A:3 A:5<Zero>

== B:4<ACK1> B:4

==

Transition2:

now

let T,N,M be natural such that

Assumptions:

```

ControlPoint(A, T) = 3 &
Contents(MBA(T)) = Message(EmptyString, ACK1) &
SOURCE_length(T) = N &
SINK_length(T) = M &
ControlPoint(B, T) = 4 &
Empty[MAB(T)];

```

```

ControlBit(Contents(MBA(T))) = ACK1
  by Assumptions, MessageDefinition;

```

```

then S1:ControlPoint(A, Nx(T)) = 4 &
  DATA_TO_BE_TRANSMITTED(Nx(T))
    = SOURCE(SOURCE_length(Nx(T))) &
  SOURCE_length(T) + 1 = SOURCE_length(Nx(T)) &
  Empty[MBA(Nx(T))] & Empty[MAB(Nx(T))]
  by A3x1, Assumptions;
then S1':ControlPoint(A, Nx{2}(T)) = 5 &
  Contents(MAB(Nx{2}(T))) =
  Message(DATA_TO_BE_TRANSMITTED(Nx{2}(T)), ZeroString)
  by A4, Assumptions;

```

then

```

  DATA_TO_BE_TRANSMITTED(Nx{2}(T))
    = DATA_TO_BE_TRANSMITTED(Nx(T)) &
  SOURCE_length(Nx(T)) = N + 1 by A4, Assumptions, S1;
then DATA_TO_BE_TRANSMITTED(Nx{2}(T))

```

```

    = SOURCE(SOURCE_length(Nx(T)))
  by S1,A4;
then Part1:ControlPoint(A, Nx{2}(T)) = 5 &
  Contents(MAB(Nx{2}(T)))
    = Message(SOURCE(N + 1), ZeroString) &
  SOURCE_length(Nx{2}(T)) = N + 1
  by S1,S1',A4,Assumptions;

ControlPoint(B, Nx(T)) = 4
  & SINK_length(Nx(T)) = M by Assumptions,B4x3;
then ControlPoint(B, Nx{2}(T)) = 4 &
  SINK_length(Nx{2}(T)) = M & Empty[MBA(Nx{2}(T))]
  by B4x3, S1;

hence ex T st
  ControlPoint(A, T) = 5 &
  Contents(MAB(T))
    = Message(SOURCE(N + 1), ZeroString) &
  SINK_length(T) = M &
  SOURCE_length(T) = N + 1 &
  ControlPoint(B, T) = 4 &
  Empty[MBA(T)] by Part1;
end;

==
== A:5<Zero>  A:5
== B:4        B:1<ACK0>
==
Transition3:
now
  let T,N,M be natural such that
  Assumptions:
    ControlPoint(A,T) = 5 &
    Contents(MAB(T)) = Message(SOURCE(N), ZeroString) &
    SOURCE_length(T) = N & SINK_length(T) = M &
    ControlPoint(B, T) = 4 & Empty[MBA(T)];

  ControlBit(Contents(MAB(T))) = ZeroString
  by Assumptions, MessageDefinition;
then S1:ControlPoint(B, Nx(T)) = 5 &
  SOURCE(N) = SINK(Nx(T), SINK_length(T)) &
  (SINK_length(Nx(T)) = SINK_length(T) + 1) &
  Empty[MBA(Nx(T))] &
  Empty[MAB(Nx(T))] by B4x1, Assumptions;
then Part1:ControlPoint(B, Nx{2}(T)) = 1 &
  SOURCE(N) = SINK(Nx{2}(T), SINK_length(Nx{2}(T))) &
  Contents(MBA(Nx{2}(T))) = Message(EmptyString, ACK0) &
  SINK_length(Nx{2}(T)) = M + 1 by B5, Assumptions;
then Part2:ControlPoint(B, Nx{2}(T)) = 1 &

```



```

SOURCE(N) = SINK(Nx{2}(T), M + 1) &
Contents(MBA(Nx{2}(T))) = Message(EmptyString, ACK0) &
SINK_length(Nx{2}(T)) = M + 1;

```

```

ControlPoint(A, Nx(T)) = 5
& SOURCE_length(Nx(T)) = N by Assumptions, A5x3;
then ControlPoint(A, Nx{2}(T)) = 5
& SOURCE_length(Nx{2}(T)) = N
& Empty[MAB(Nx{2}(T))] by A5x3, S1;

```

hence ex T' st

```

ControlPoint(B, T') = 1 &
SOURCE(N) = SINK(T', M + 1) &
Contents(MBA(T')) = Message(EmptyString, ACK0) &
SINK_length(T') = M + 1 &
SOURCE_length(T') = N &
ControlPoint(A, T') = 5 &
Empty[MAB(T')] by Part2;

```

end;

==

== A:5 A:3<One>

== B:1<ACK0> B:1

==

Transition4:

now

let T,N,M be natural such that

Assumptions:

```

ControlPoint(A, T) = 5 &
Contents(MBA(T)) = Message(EmptyString, ACK0) &
SOURCE_length(T) = N &
SINK_length(T) = M &
ControlPoint(B, T) = 1 &
Empty[MAB(T)];

```

```

ControlBit(Contents(MBA(T))) = ACK0
by Assumptions, MessageDefinition;
then S1:ControlPoint(A, Nx(T)) = 2 &
DATA_TO_BE_TRANSMITTED(Nx(T))
= SOURCE(SOURCE_length(Nx(T))) &
SOURCE_length(T) + 1 = SOURCE_length(Nx(T)) &
Empty[MBA(Nx(T))] &
Empty[MAB(Nx(T))] by A5x1, Assumptions;
then S1':ControlPoint(A, Nx{2}(T)) = 3 &
SOURCE_length(Nx{2}(T)) = N + 1 &
Contents(MAB(Nx{2}(T))) =
Message(DATA_TO_BE_TRANSMITTED(Nx(T)), OneString)
by A2;
then

```

```

DATA_TO_BE_TRANSMITTED(Nx{2}(T))
  = DATA_TO_BE_TRANSMITTED(Nx(T)) &
SOURCE_length(Nx(T)) = N + 1
  by A2, Assumptions, S1;
then DATA_TO_BE_TRANSMITTED(Nx{2}(T))
  = SOURCE(SOURCE_length(Nx(T))) by S1, A2;
then Part1: ControlPoint(A, Nx{2}(T)) = 3 &
SOURCE_length(Nx{2}(T)) = N + 1 &
Contents(MAB(Nx{2}(T)))
  = Message(SOURCE(N + 1), OneString)
  by S1, Assumptions, A2, S1';

ControlPoint(B, Nx(T)) = 1 &
SINK_length(Nx(T)) = M by Assumptions, B1x3;
then ControlPoint(B, Nx{2}(T)) = 1 &
SINK_length(Nx{2}(T)) = M &
Empty[MBA(Nx{2}(T))] by B1x3, S1;

hence ex T' st
ControlPoint(A, T') = 3 &
SOURCE_length(T') = N + 1 &
Contents(MAB(T'))
  = Message(SOURCE(N + 1), OneString) &
ControlPoint(B, T') = 1 & SINK_length(T') = M &
Empty[MBA(T')] by Part1;
end;

PrefixBasis:
now
  let T be natural such that
  Assumptions:
    prefix[0, T] &
    ControlPoint(A, T) = 1 &
    ControlPoint(B, T) = 1 &
    SOURCE_length(T) = 0 &
    SINK_length(T) = 0 &
    Empty[MBA(T)] &
    Empty[MAB(T)];

consider T2 such that
Step2:
  ControlPoint(A, T2) = 3 &
  SOURCE_length(T2) = 0 + 1 &
  Contents(MAB(T2))
    = Message(SOURCE(0 + 1), OneString) &
  ControlPoint(B, T2) = 1 &
  SINK_length(T2) = 0 &
  Empty[MBA(T2)] by Assumptions, Transition0;

consider T3 such that

```

Step3:

```

ControlPoint(B, T3) = 4 &
SOURCE(0 + 1) = SINK(T3, 0 + 1) &
Contents(MBA(T3))
  = Message(EmptyString, ACK1) &
SINK_length(T3) = 0 + 1 &
SOURCE_length(T3) = 0 + 1 &
ControlPoint(A, T3) = 3 &
Empty[MAB(T3)] by Step2, Transition1;

```

prefix[0 + 1, T3] by Step3, Assumptions, PrefixForBasis;

hence ex T' st

```

prefix[0 + 1, T'] &
ControlPoint(B, T') = 4 &
Contents(MBA(T')) = Message(EmptyString, ACK1) &
ControlPoint(A, T') = 3 &
Empty[MAB(T')] by Step3, Assumptions;

```

end;

now

let T, N be natural such that

Assumptions:

```

prefix[N, T] &
ControlPoint(B, T) = 4 &
Contents(MBA(T)) = Message(EmptyString, ACK1) &
SINK_length(T) = N &
SOURCE_length(T) = N &
ControlPoint(A, T) = 3 &
Empty[MAB(T)];

```

consider T2 such that

Step2:

```

ControlPoint(A, T2) = 5 &
Contents(MAB(T2))
  = Message(SOURCE(N + 1), ZeroString) &
SINK_length(T2) = N &
SOURCE_length(T2) = N + 1 &
ControlPoint(B, T2) = 4 &
Empty[MBA(T2)] by Assumptions, Transition2;

```

consider T3 such that

Step3:

```

ControlPoint(B, T3) = 1 &
SOURCE(N + 1) = SINK(T3, N + 1) &
Contents(MBA(T3)) = Message(EmptyString, ACK0) &
SINK_length(T3) = N + 1 &
SOURCE_length(T3) = N + 1 &
ControlPoint(A, T3) = 5 &
Empty[MAB(T3)] by Step2, Transition3;

```

consider T4 such that

Step4:

ControlPoint(A, T4) = 3 &

SOURCE_length(T4) = (N + 1) + 1 &

Contents(MAB(T4))

= Message(SOURCE((N + 1) + 1), OneString) &

ControlPoint(B, T4) = 1 &

SINK_length(T4) = N + 1 &

Empty[MBA(T4)] by Step3, Transition4;

consider T5 such that

Step5: ControlPoint(B, T5) = 4 &

SOURCE((N + 1) + 1) = SINK(T5, (N + 1) + 1) &

Contents(MBA(T5)) = Message(EmptyString, ACK1) &

SINK_length(T5) = (N + 1) + 1 &

SOURCE_length(T5) = (N + 1) + 1 &

ControlPoint(A, T5) = 3 &

Empty[MAB(T5)] by Transition1, Step4;

S3: SINK(T3, N + 1) = SOURCE(N + 1) by Step3;

S5: SINK(T5, (N + 1) + 1) = SOURCE((N + 1) + 1)
by Step5;

prefix[((N + 1) + 1), T5]

by PrefixInduction, Assumptions, S3, S5;

hence ex T' st prefix[((N + 1) + 1), T'];

end;

end

Appendix A6 Syntax of PDL

This appendix describes the syntax of the Protocol Description Language using the Backus Normal form (BNF) notation. The syntax of Protocol Description Language is explained in Chapter 3.

```
<protocol> ::= protocol <name>
               configuration
                 <channel description> +
               end_configuration
                 <party definition> +
               end_protocol <name>

<name> ::= <characters> +

<name series> ::= <name> [ , <name series> ]

<channel description> ::= channel <name> <party access> +

<party access> ::= <party name> <access type>;

<access type> ::= transmit | transmit-receive | receive

<party definition> ::= <party body> | <party repetition>

<party body> ::= party <party name>
               [ in_client <name series> ; ]
               [ out_client <name series> ; ]
               [ local <name series> ; ]
               party_init
                 <statement series>
                 <next state>
               end_party_init
                 <state> +
               end_party <party name>

<statement series> ::= <statement> [ ; <statement series> ]

<statement> ::= <start timer> | <stop timer> | <transmit>
               | <read statement> | <write statement>

<start timer> ::= start_timer;

<stop timer> ::= stop_timer
```

```

<transmit> ::= transmit on <name> <transmit string> ;
<transmit string> ::= <expression> | <built in function>
<expression> ::= <term> | <term> || <expression>
<term> ::= "<character>+" | <variable name>
<built in function> ::= <name> ( [<name series>] )
<read statement> ::= read_client( <client name>, <variable name> )
<write statement> ::= write_client( <client name>, <variable name> )
<next state> ::= next_state <name> ; | <return>
<return> ::= return(SUCCESS); | return(FAILURE);

<state> ::=
    state <name>
    state_init
        <statement series>
    end_state_init
    wait_event
    <event> +

    unspecified
        <statement series>
        <next state option>
    end_state <name>

<event> ::=
    receive
        <text or timeout> ;
        <statement series>
        <next state option>
    end_receive

<text or timeout> ::= <text> | <timeout>
<text> ::= on <name> <expression>
<timeout> ::= timeout
<next state option> ::= <next state> | next_state same
<party repetition> ::= party <party name> same_as <party name>;

```