PROTOCOL VERIFICATION USING SYMBOLIC MODEL CHECKING

by

CHARLES G. MATHIESON

B.Sc., The University of British Columbia, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming

THE UNIVERSITY OF BRITISH COLUMBIA

DECEMBER 1993

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date JANUARY 14, 1994

DE-6 (2/88)

Abstract

To reduce problems encountered in the later phases of the software life cycle, verification techniques can be used in the design phase to ensure that a design has the intended properties. The main advantage of using formal verification over other validation methods, such as simulation and testing, is that it reasons about all possible behaviours of a system. However, formal verification techniques have not yet been widely accepted in industry because most of them suffer from the state explosion problem or are too difficult to use.

In this thesis, an automatic model checking verification system for communication protocols is developed that tackles the state explosion problem. The Ever symbolic verifier [HDDY92], which is a high-level specification language and symbolic reachability analysis tool for extended finite state machines, is used as a basis for this system. The system accepts a protocol specification written in Estelle.Y [Lu91], a variant of the Estelle formal protocol description language [ISO89]. Each Estelle.Y module of the specification is translated into the Ever language and fed into the Ever symbolic verifier. The intended properties are written as CTL temporal logic formulae [CG87] expressed in terms of variables in the Estelle.Y specifications. These formulae are given to the verifier to

check that they are true in the model produced from the protocol specification. With this system one can assert that given safety and liveness properties are true in all possible behaviours of a protocol. When the system finds a given formula to be false, it is capable of producing a counterexample trace. This trace greatly assists the designer to correct the protocol specification and the temporal formulae of the intended properties.

After the Estelle.Y to Ever translator was implemented, the original Ever verifier was extended to support model checking of CTL temporal formulae [CG87]. The extended verifier can check not only eventuality properties but also more general temporal properties such as precedences and invariances. The Ever verifier was also extended with the notion of fairness constraints [BCMD90] to allow the model to check only fair behaviours. These extensions enable incremental verification to be performed to reduce the overall checking time dramatically.

This system was successfully applied to the specification of the alternating bit protocol [ISO89] to demonstrate this new tool. Various safety and liveness properties expressed as CTL temporal formulae are described and explained in detail in this thesis. The CPU time and memory requirements for this verification are discussed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I wish to sincerely thank my co-supervisors, Dr. Samuel Chanson and Dr. Son Vuong, for their invaluable guidance and support given throughout my research work.

I wish to acknowledge the partial financial support from the Canadian Institute of Telecommunications Research and the Department of Computer Science of UBC.

I wish to thank the staff of the Computer Science department for providing an environment most suitable for carrying out my research.

I wish to acknowledge R.A.C.E. Technologies, Inc., where I worked in the first three years of my part-time studies, for providing a work environment that gave me the inspiration to find the interesting research topic covered in this thesis.

Chapter 1: Introduction

1.1. Motivation and Objectives

The goal of this thesis is to develop a tool that will increase the likelihood that an implementation of a communication protocol will carry out its operations in exactly the way intended.

Often bugs arise in software as a result of misunderstandings between specifiers and implementors. To resolve this problem, formal description techniques should be used to ensure all users of a specification interpret it in exactly the same way.

The main objective of this thesis is to develop a validation tool that rigorously determines whether a specification will behave in the manner intended. The designer will write the specification in a formal protocol description language and write intended safety and liveness properties as temporal logic formulae. This system will determine whether a given temporal property is true in the system corresponding to the specification. If it determines the property to be false, a counterexample will be produced to assist the designer to find the cause of the inconsistency between the specification and the formula. The designer then can modify the specification

appropriately and repeat this process until conformity is reached.

This tool is not intended to determine the performance of protocols. It is to ensure the protocol itself, regardless of the implementation's performance, conforms to its specification. The verification will reason about all possible timing behaviours of the specification.

1.2. Thesis Contributions

An automatic verification tool, based on the Ever verifier [HDDY92], was developed that checks whether a given Estelle.Y [Lu91] protocol specification satisfies given temporal properties written as temporal logic formulae. The Estelle.Y language was chosen because it is a variant of the Estelle standard protocol description language and is used for other protocol development tools at UBC. Ever was used because its command language is most suited for translation from Estelle.Y and it provides the best foundation upon which to build a CTL model checker.

The BDD based Ever verifier [HDDY92,Hu92-93] was extended to implement CTL temporal logic model checking with fairness constraints. This component is the main part of the system for verifying that a temporal logic specification matches an extended state machine description. An improvement was made to

the verifier to make outputting of propositions meaningful to the user (i.e. printing symbolic names instead of numerical addresses of BDD variables).

A translator was implemented to automate the entry of Estelle.Y protocol specifications into the verifier.

A new approach for carrying out incremental verification of Estelle.Y specifications that does not suffer from the state explosion problem has been developed. The CTL model checker with fairness constraints is used. A nextstate relation for the behaviour of the Estelle.Y modules is produced by the translator mentioned above. A very simple nextstate relation for the surrounding environment is manually written in Ever code. Then fairness constraints are used to restrict the set of possible behaviours considered during verification. This method has successfully been applied to the alternating bit protocol.

1.3. Thesis Outline

In chapter 2, the major and common protocol verification methods are discussed with reasons for choosing the symbolic model checking approach. A number of existing tools that were considered are compared. The reasons for using the Ever verifier [HDDY92] are given.

In chapter 3, the theory of symbolic model checking upon which the Ever verifier is based is reviewed to give sufficient background theory for chapter 4.

In chapter 4, the implementation of the Estelle.Y to Ever translator, *estelle2ever*, and the CTL model checking extensions to the Ever verifier are explained. The *estelle2ever* translator translates a set of Estelle.Y modules describing a system into a set of Ever propositions. The mappings from Estelle.Y commands and statements to Ever propositions are explained.

The way the CTL symbolic model checking algorithms discussed in chapter 3 were integrated into the Ever verifier is described. A number of other improvements were made to the Ever system. These include printing meaningful names for referenced variables in displayed expressions, the addition of two new Ever commands (*deffreevar* and *setdefaultnextstate*). The purpose of these two new commands is given while discussing the model checking implementation.

In chapter 5, this verification tool is applied to the alternating bit protocol. A strategy is proposed to prove the correctness of the protocol by showing that a number of internal properties are true and giving an argument as to why the protocol is correct when these internal properties are true. Additional internal properties specifying that data in the data packets are never modified in transit are verified to show that the control structure of the protocol is valid for all data

packet sizes. The quantities of CPU time and memory used to verify all these properties are discussed. A method for producing counterexample traces is proposed and demonstrated with an example from the alternating bit protocol.

In chapter 6, some conclusions are drawn and some optimizations that should be implemented are discussed.

The appendices contain listings of files used throughout the verification of the alternating bit protocol. Appendices 1 and 2 contain source code for the specification in Estelle.Y and ASN.1, respectively. The Ever code produced by the *estelle2ever* translator is shown in Appendix 3. The Ever script file containing the commands for carrying out the verification is in Appendix 4. The final verification output is shown in Appendix 5. Commands and output for verification of a formula corrected after producing a counterexample trace, shown in Appendix 6, are in Appendices 7 and 8, respectively.

Chapter 2: Overview of Verification Approaches and Tools

In Section 2.1, a few common protocol verification techniques are reviewed, compared and contrasted with reasons for choosing symbolic model checking.

In Section 2.2, the features of a number of tools considered for use as a basis for the research conducted in this thesis are briefly described. The reasons for choosing the Ever verifier are given.

Chapter 3 reviews the theories of symbolic model checking upon which the implementation described in chapter 4 is based.

2.1. Protocol verification methods

2.1.1. Model checking

Model checking refers to a method for determining whether a given structure (e.g. finite state machine) is a model of a given temporal formula.

The first model checking algorithms explicitly represent the state space of the model being checked. Clarke and Grumberg review the history of temporal logic verifiers in [CG87]. Clarke and Emerson [CE81] designed the first automatic model checking verifier for CTL temporal logic. The performance was

polynomial in both the size of the model and the length of the temporal logic formula. Later, Clarke, Emerson and Sistla devised an improved CTL model checking algorithm [CES86] with complexity linear in the product of the length of the formula and the size of the global state graph.

They added an extension to the algorithm to make it check only *fair computations* without any additional complexity expense because otherwise many formulae would be false in the model. For example, we are not interested in including the case where a data link between two systems is permanently disconnected. Gabbay et al. were one of the first to introduce the concept of fairness to temporal logic [CES86, GPSS80]. (When verifying a property of a system of concurrent processes, we wish to only consider execution sequences in which all processes execute infinitely often.)

These algorithms check whether a given temporal logic formula is true in a given labelled transition system. A labelled transition system is a finite state machine graph with each node having a label containing a list of formulae which are true when the system is in that state. Initially the labels in the graph only contain atomic propositions. The algorithm works in stages. In the first stage, all subformulae of length 1 in the original formula are checked for truth in all the graph nodes. In the second stage, subformulae of length 2 are checked, and so on, until the original formula is checked. Each

time a subformula is checked for truth at a node, an appropriate set of other related nodes (as defined by the top level operator of the subformula) are checked to determine the truth of the subformula at this node. For example, the EX next time operator of CTL temporal logic requires any one of the immediate successors of the node to have the EX operator's operand true. When the algorithm terminates, the original formula is true in all nodes that contain it in their labels.

Sistla and Clarke analyzed the model checking problem for a variety of temporal logics and showed the problem is PSPACE complete for linear temporal logic [SC86].

Clarke and Grumberg introduced an algorithm for model checking concurrent systems with many similar processes [CG89].

2.1.2. Symbolic Model Checking

Burch, Clarke, McMillan and Dill [BCMD90] have extended the temporal logic model checking algorithm by Clarke, Emerson and Sistla [CES86] to represent the state graph with binary decision diagrams [Bry86] instead of with an explicit labelled transition system. This BDD represents a predicate transformer from the current state to the next state. This avoids the explicit construction of the state graph. (Burch et al. acknowledge that Bose and Fisher [BF89] described a binary decision diagram (BDD) based algorithm for CTL model checking without support for fairness constraints.) For state spaces with some regularity,

this representation is often more efficient. Thus systems with extremely large numbers of states can be verified with this algorithm. They have demonstrated it with a specification of a synchronous pipelined design with approximately 5×10^{20} states.

Instead of representing the relationship between the current and the next states in a labelled transition system explicitly with a state graph, it is represented as a Boolean function of all the variables in the current and next states. For each possible combination of variable values in the current and next states, this function indicates whether a transition is possible (in the labelled transition system) from this current set of variable values to this next set of variable values.

Bryant defined algorithms for carrying out basic operators on BDDs such as Boolean connectives (i.e. *and*, *or*, *not*), functional composition, computing restrictions for functions (i.e. substituting a constant for one of its arguments), and quantification over Boolean variables. These basic operations are used for computing values of state formulae (i.e. temporal logic formulae without any temporal operators).

To compute the values of path formulae (i.e. those with temporal operators), Burch et al. devised new algorithms for computing values of temporal formulae in binary decision diagrams. Given a temporal formula f, and a transition relation R (represented as a BDD in the first version of the algorithm), this new algorithm computes the set of current variable values

(a new BDD) for which the temporal formula f[1] is true in the labelled transition system associated with R. These algorithms are based on fixpoint characterizations of CTL temporal logic operators [BCMDH90, BCMD90]. A fixpoint of a function is a value at which the function applied to the value is the value itself (i.e. $f(x)=x$) [Tar55]. These fixpoint characterizations are explained in more detail later.

Clarke et al's explicit version of the model checking algorithm allows fairness constraints to be only state formulae (i.e. formulae without temporal operators). However, the symbolic model checking algorithm [BCMD90] allows fairness constraints to be arbitrary CTL formulae. According to Burch [Bur93], these are equivalent.

Burch et al. suggest that their algorithm is the first BDD based model checking algorithm for CTL temporal logic for non-deterministic labelled transition systems [BCMD90].

2.1.3. Protocol projections

In an attempt to prevent the state explosion problem that can easily occur in reachability analysis, Lam and Shankar [LS82] devised a method of verification with protocol projections. Real life protocols typically have several distinguishable functions. To verify one of these functions,

---

[1]The formula *f* is assumed to only refer to variables in the current phase.

one is to construct an *image protocol* that just specifies this one function in such a way as to satisfy a *well-formed* property. The complexity of the image protocol is shown to always be less than that of the original protocol. They show that if the image protocol satisfies the well-formed property then any safety and liveness properties that are valid in the image protocol are also valid in the original protocol. Thus with this method, one can verify properties in protocols that are larger than those that can be verified with reachability analysis alone.

This method was not considered because it appears the procedure for constructing image protocols cannot be easily automated.

2.1.4. Theorem proving

Theorem proving is the most powerful method to carry out protocol verification because it is not constrained by the size of the specification. However, it probably is the most tedious because it requires a lot of human ingenuity to actually carry out the proof successfully [Pn1, ES1]. With an automatic theorem prover, typically most of the proof steps are generated automatically, but the crucial decisions on proof strategy have to be made manually.

The claim to be proven is specified as a formula in the logic being used. The implementation of the protocol is translated into a series of assertions (formulae in the logic).

The set of axioms and inference rules of the logic are combined with the assertions to carry out a proof that the claim is true. One repeatedly uses the axioms and inference rules to derive the claim from the assertions.

For example, Manna and Pneuli describe a global proof system for proving propositional temporal logic properties in transition based implementations [Pn1,MP4]. The proof system has three parts:

- The *general part* has proof methods for general temporal logic formulae expressed in terms of atomic proposition variables. Axioms, inference rules and tableau methods are used here for checking the validity and satisfiability of general temporal logic formulae.
- The *domain part* uses axioms and inference rules of theories of data structures of variables (in the claim specification and implementation specification) (e.g. Booleans, integers, lists, sets) to prove assertions about these data structures. Inductive schemes are used to prove properties that apply to all sizes or values of these data types.
- The *program part* transforms the implementation specification into a set of assertions expressed in the temporal logic. Atomic propositions in these assertions may be expressed as Boolean expressions of data structures used

in the program. These assertions are treated as assumptions (axioms) during the proof of the claim.

Manna and Pnueli have defined an *invariance rule* and a few *liveness rules* for proving safety and liveness properties, respectively [Pn1,MP5].

This method was not chosen because one cannot generally develop a system that carries out the proofs automatically. The proofs most often involve a lot of human expertise.

2.1.6. Simulation

These systems translate a formal Estelle specification into an implementation [CLV93]. Then a simulator (interpreter) is used to execute the implementation in a debugging environment where variables, queues and module instance creations and destructions can be observed. Only a single execution path can be tested at a time with this approach.

To test whether an implementation satisfies a given temporal formula, heuristics could be used to automatically select appropriate paths for testing. There is no guarantee when a finite number of tests have been performed that there does not exist an untested path that does not satisfy the temporal formula.

This approach was not chosen because it does not attempt to cover all possible execution paths in the implementation.

2.1.7. Symbolic evaluation

Clarke and Richardson describe three methods of symbolic evaluation of programs in [CR85]. First, *path dependent symbolic evaluation* analyzes a single given path through a routine. A symbolic expression in terms of the input variables for each output variable is derived. Clearly this method is not appropriate for verifying protocol specifications because there are just too many (possibly infinite) paths to check.

Second, they describe *dynamic symbolic evaluation* which derives symbolic expressions for all output variables along the path determined by a given set of input variable values. To apply this method to protocol verification, one would have to apply it to every possible set of input values. Clearly, this is not appropriate.

Third, they describe the *global symbolic evaluation* method. The goal here is to derive a global representation of a routine for all possible execution paths through the routine. Initially, input parameters are assigned symbolic names and internal variables are assigned constant values. As each program statement is processed, the symbolic expressions for the output variables are updated according to rules defined by the semantics of the current program statement. For example, when an assignment statement is processed, all references in current output expressions to the assigned variable are updated with the expression from the assignment statement.

The main limitation of this approach is the problem of deriving expressions for arbitrary loops. It tries to replace each loop, which can represent infinite paths, with a closed-form expression that captures the effect of the loop. A symbolic expression for each output variable in terms of the state at the beginning of the loop has to be derived. (Here we briefly describe their loop analysis technique.) In order to do this, a conditional expression representing a *final iteration count* expressed in terms of symbolic values of variables at entry to the loop has to be determined. Also, for each variable modified by the code of the loop, the symbolic value at exit from the loop must be expressed in terms of both the final iteration count and the symbolic values of variables at entry to the loop. Recurrence relations for all variables modified in the loop are derived. A loop exit condition in terms of the kth iteration of loop variable values is derived. Attempts are made to solve the recurrence relations to forms in terms of variables before entry to loop. Obtaining solutions for recurrence relations is not always straightforward and is sometimes impossible (e.g. interdependence between two recurrence relations, conditionals within loop exacerbate problem, exit conditions in terms of conditional expressions or minimum value expressions, etc.)

This method seems to be suited for traditional programs that process input data and terminate with output data. Protocol

implementations tend to be programs that run continuously. This method was not used because it does not reason about the state of a program from one time unit to the next.

2.2. Tools considered

A brief description of the advantages and disadvantages of each of following tools considered is given. The Ever tool was chosen because it provides a language for describing high level data structures and provides symbolic reachability analysis.

2.2.1. LITE Tableau verifier

The LITE Tableau verifier package was considered. It implements a tableau construction algorithm in Prolog for interval temporal logic. When given an interval temporal logic formula it generates a compatible state machine. If this tool was to be used, an algorithm would have to be developed to compare the state machine from the protocol specification with the one produced by the LITE tableau verifier. The problem of comparing automata is generally more complex than model checking. Thus this tool was not chosen.

The advantage of using this tool would be that it reasons about interval temporal logic. This would be good for reasoning about the performance of protocols because this logic can express explicit discrete time intervals.

2.2.2. Murphi

Murphi is a language and system for defining and verifying the behaviour of a system. A system is specified as a set of rules. I found by doing some trivial experiments that Murphi does its verification with explicit reachability analysis. No symbolic evaluation is used.

Murphi provides no means for expressing properties with temporal operators. It can only express properties that are written in terms of only the current state of the system.

2.2.3. VOSS

VOSS is an interval temporal logic symbolic model checker with a hardware oriented specification language. The specification language is too clumsy for the purpose of translating protocol specifications in this thesis.

2.2.4. Ever

Ever is a binary decision diagram (BDD) based verifier with a high-level description language for defining data types, variables, propositions and predicates.

It includes a routine for determining reachable states. One defines the behaviour of a system by transforming a high level language description into a nextstate relation. By avoiding the

complete evaluation of the nextstate relation, Ever efficiently implements a reachability analysis algorithm.

This tool was chosen because it provides a specification language most suited for translation from Estelle.Y and provides a good foundation for implementing symbolic model checking algorithms.

2.2.5. Hol

Hol is a very powerful theorem proving system based on high order logic. However, it is very difficult to do proofs in this system. The proofs cannot be completely automated. They require a lot of human intelligence to carry out. This method was not chosen because an automatic verification system is desired.

2.2.6. EDT

EDT (Estelle Development Toolkit) is a simulation package for Estelle specifications. It allows one to watch the execution of an Estelle specification with controlled or random input data. Various objects may be monitored such as variables, interaction queues, and creations and destructions of module instances.

This tool is not appropriate for verification since it is not designed for reasoning about all possible execution paths.

2.2.7. SMV

SMV is a CTL BDD based symbolic model checker with a specification language only at the bit level.

It provides a limited notion of fairness. The only type of fairness supported is that all processes in a concurrent system are assumed to execute infinitely often. A user may wish to specify more specific types of fairness (e.g. that a message is sent by a sender infinitely often). Perhaps, this type of fairness is sufficiently powerful to describe general fairness requirements.

This tool was not chosen because its specification language is only at the bit level.

2.3. Summary

The symbolic model checking verification method [BCMD90,BCMDH90] was chosen because of its potential for reasoning about extremely large systems automatically.

The Ever verifier [HDDY92] was chosen because it provides the best foundation upon which a symbolic model checking system could be developed. It also provides a high-level description language most suitable for translating Estelle.Y [Lu91] protocol specifications.

The theory of CTL model checking is presented in the next chapter to provide sufficient background material for the implementation described in chapter 4.

Chapter 3: Theoretical Background

In this chapter, the fundamental theories upon which this verification system's implementation, described in chapter 4, is based are explained. Algorithms and data structures of two main CTL model checking approaches are discussed.

In section 3.1, the original explicit state space model checking algorithm is explained. This algorithm uses a labelled transition system data structure that explicitly represents each state with a node in the graph structure. Associated with each node is a label containing a list of atomic propositions true in the corresponding state. The model checking algorithm recursively evaluates subformulae of the given temporal formula by searching for nodes satisfying constraints defined by the subformulae operators.

Section 3.2 explains the symbolic version of the model checker. This includes a brief description of binary decision diagrams which are used to represent binary functions. The state space of a system is modelled with a binary function of all variables in the current and next states indicating whether a transition is possible from that given current state to that given next state. Temporal operators in temporal formulae are evaluated using the fixpoint characteristics of the CTL temporal logic. The notion of fairness constraints is explained with appropriate extensions to the model checking algorithm.

3.1. Explicit CTL Model Checking

Section 3.1.1 defines the data structures used by the model checking algorithm. Section 3.1.2 reviews a few different types of temporal logic and explains the advantages to using the CTL temporal logic. Sections 3.1.3 and 3.1.4 describe the algorithm without and with fairness constraints supported, respectively.

3.1.1. Kripke structures for labelled transition systems

All the temporal logics discussed below use the Kripke model of labelled transition systems for representing the system in which a given temporal logic formula is checked. The older versions of the model checking algorithms use the Kripke model explicitly [CES86]. The newer version of the CTL model checking algorithm represents the program with a binary decision diagram (BDD) that relates the current state with the next state. This BDD corresponds to a Boolean function of all program variables in the current and next states [BCMD90].

The labelled transition system is formally defined as a 3-tuple:

M = (S,R,L) where

S	is a set of states
$R \subseteq S \times S$	is the transition relation (determining between which pairs of states transitions exist). R must be total, in other words, there must be at least one transition from each state $s \in S$.

$L: S \rightarrow 2^{AP}$	Associated with each state is a list of atomic propositions which are true in that state. Note the absence of an atomic proposition implies that it is false in the state. This assumption is different from the interpretation in the new model.

We define the notation $S_1 \rightarrow S_2$ to indicate $(S_1,S_2) \in R$, in other words, that there is a transition from s_1 to s_2 in the model.

We define a path in the model M to be a sequence of states $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $s_i \rightarrow s_{i+1}$.

We use π^i to denote the suffix of π starting at s_i.

3.1.2. CTL* Logic

Temporal logic provides a formal system for describing and reasoning about the occurrence of events in time.

This logic has the regular propositional logic operators, a number of temporal operators and two path quantifier operators.

The temporal operators allow the expression of properties of temporal systems such as *invariances* (properties that are always true), *eventualities* (properties that must becomes true at some future instant), and *precedences* (properties that state that one event must occur before another).

The existential quantification operator indicates whether the given property must occur on one possible execution path from the current state.

The universal quantification operator indicates that the given property must occur on all possible execution paths from the current state.

The set of CTL* (computation tree logic) formulae is defined by the following formal rules. CTL* has 2 types of formulae: state formulae (which are true in a given state) and path formulae (which are true along a specific path).

Let AP be the set of atomic propositions.

state formulae:

A, if $A \in AP$

If f and g are state formulae, then $\neg f$ and $f \vee g$ are state formulae.

if f is a path formula, then E(f) is a state formula.

Path formulae:

If f is a state formula, then f is a path formula.

If f and g are path formulae, then $\neg f$, $f \vee g$, Xf, f U g are path formulae.

The semantics of a CTL* formula are defined with respect to a Kripke structure representing the program. The standard notation to indicate that a state formula f is true in a structure M is $M,s \models f$. This means that formula f is true at state s in structure M. When M is understood from the context, we simply write $s \models f$.

The *holds* $\models$ relation is defined inductively as follows:

Assume f1, f2 are state formulae, g1,g2 are path formulae.

1. $s \models A \Leftrightarrow A \in L(s)$. { atomic proposition A in label at state s }

2. $s \models \neg f1 \Leftrightarrow \neg(s \models f1)$

3. $s \models f1 \vee f2 \Leftrightarrow s \models f1$ or $s \models f2$

4. $s \models E(g1) \Leftrightarrow$ for some state t such that $(s0,t) \in R$, $t \models g1$

5. $\pi \models f1 \Leftrightarrow$ s is the first state of path π and $s \models f1$ (says that a state formula is true on a given path iff the state formula is true in the first state of the path)

6. $\pi \models \neg g1 \Leftrightarrow \neg(\pi \models g1)$.

7. $\pi \models g1 \vee g2 \Leftrightarrow \pi \models g1$ or $\pi \models g2$

8. $\pi \models Xg1 \Leftrightarrow \pi^1 \models g1$.

9. $\pi \models g1\ U\ g2 \Leftrightarrow$ there exists $k \geq 0$ such that $\pi^k \models g2$ and for all $0 \leq j < k$, $\pi^j \models g1$

The following abbreviations are used:

$f \wedge g \equiv \neg(\neg f \vee \neg g)$

$Ff \equiv true\ U\ f$

$A(f) \equiv \neg E(\neg f)$

$Gf \equiv \neg F \neg f$.

These abbreviations are a consequence of duality properties of the propositional logic and CTL* temporal logic.

CTL* is the most expressive of the temporal logics presented here.

Two other temporal logics often discussed in the literature are LTL (linear temporal logic) and CTL (computation tree logic) which are both subsets of CTL* with less expressive power.

The set of valid LTL formulae is defined by the following rules:

A state formula is:

Af where f is a path formula.

A path formula is:

A if A $\in$ AP (i.e. an atomic proposition)

If f and g are path formulae, then $\neg$f, f $\vee$ g, Xf, f U g are path formulae.

CTL is the subset of CTL* restricting the use of path quantifiers to be combined with temporal operators X, U, G, and F. In other words, each path quantifier must be immediately followed by a temporal operator.

Formally, the state formulae rules of CTL are the same as those of CTL*.

The path formulae are limited to the following:

If f and g are state formulae, then Xf and f U g are path formulae.

If f is a path formula, then $\neg$f is a path formula.

Branching time temporal logics can express all the properties expressible in linear time logic and more [ES88].

Note, LTL formulae have no path quantifiers, thus all LTL formulae must state properties that are true on all paths in the model.

The model checking problem for linear temporal logic and CTL* is exponential in the length of the formula and linear in the size of the global state graph [LP85][CG87].

Clarke and Grumberg [CG87] show that the model checking algorithm for CTL is linear in the number of states and the number of transitions and is linear in the length of the formula.

Since the complexity of the model checking problem for CTL is exponentially less than that of CTL* and LTL, and the expressive power of CTL seems more flexible than LTL, CTL was chosen. CTL allows one to describe properties that must apply on all possible execution paths or that must only apply on at least one path. LTL only allows properties that must apply on all paths to be expressed.

3.1.3. Explicit enumerative model checking algorithm

The model checking algorithm for CTL formulae in a Kripke model of a system [CES86] is described below.

The goal of the algorithm is to determine whether there exists a state in a given Kripke structure M such that a given CTL temporal formula f is true (in that state).

The steps of the algorithm are as follows:

1. Normalize formula using duality properties of CTL logic to have formula expressed with only the operators given in earlier definitions (i.e. EX, EU, $\neg$, $\vee$). The duality properties are:

$$EFf \equiv E[True\ U\ f]$$

$$EGf \equiv \neg EF\neg f$$

$$AXf \equiv \neg EX\neg f$$

$$A[f\ U\ g] \equiv \neg E[\neg g\ U\ (\neg f \wedge \neg g)] \wedge \neg EG\neg g$$

$$AGf \equiv \neg EF\neg f \equiv \neg E[True\ U\ \neg f]$$

$$AFf \equiv A[True\ U\ f]$$

2. Logically construct a parse tree for formula f (with operators at nodes and atomic propositions in leaves).

3. Associate with each state in graph M = (S, R, L) a list of subformulae of f which are true at that state. Initially set all these lists to the empty list.

4. Process all subformulae g of f, processing them in order of increasing length (i.e. all length 1 subformulae, then length 2 subformulae, and so on). For each subformula g, determine for each state $s \in S$ whether the subformula is true in state s. If so, add subformula g to label at state s. The algorithms for determining the truth of subformulae at states are briefly described below.

When iterative step 4 is finished, labels of all states in which original formula f is true will contain the original formula f.

In each iteration of step 4 above, an appropriate search of part of the graph and parse tree is performed to determine the truth of the current subformula. The part of the graph searched corresponds to the descriptions of the operators in the following table.

atomic proposition	true if same atomic proposition in state's label.
¬f	true if f not in state's label.
f1 ∨ f2	true if either f1 or f2 in state's label.
EXf	true if there is at least one successor state of s containing f in its label.
AXf	true if all successor states of s contain f in their labels.
E[f1 U f2]	true if there is at least one path with a state containing a label with f2 and which has f1 in labels of all previous states on path.
A[f1 U f2]	true if all paths lead to a state with f2 in its label with all previous states containing f1.

3.1.4. Fairness constraints extension

Often during verification of concurrent systems such as communication protocols, we are only interested in considering *fair* execution paths while model checking. [CES86] defines a fair path as one where a set of *fairness constraints* occur infinitely often. For example, when verifying a protocol implemented as two modules connected by an unreliable data link,

we are not interested in considering cases of when the data link continuously loses data or is disconnected. Without using fairness constraints, cases like this example would cause formulae we expect to be true under "normal conditions" to end up being false.

Clarke et al. developed an extension to their model checking algorithm to handle fairness without any additional complexity (time and memory).

The model is extended from a 3-tuple to a 4-tuple M=(S,R,L,F) with the first three members the same as before and F being a collection of state predicates (i.e. $F \subseteq 2^S$). A path p is *F-fair* iff

> for each $g \in F$, there are infinitely many states on p which satisfy predicate g.

Since we have used a symbolic model checking algorithm, details of the explicit version of the fairness algorithm are not given here. See [CES86].

3.2. Symbolic CTL Model Checking

The previous section explained the original version of the CTL model checking algorithm which uses a model that explicitly enumerates all the states of the implementation being checked.

Burch et al. [BCMD90,BCMDH90] have devised a new improved model checking algorithm that does not enumerate all the states explicitly, but instead represents the state space as a

nextstate relation stating how all states in current time unit are related to all states in the next time unit. Bryant's binary decision diagrams [Bry86] are used to represent Boolean functions.

A number of other authors have developed other BDD based verification systems for deterministic systems such as combinational circuits [CBM89].

Burch et al. appear to be the first to have come up with a BDD based model checking algorithm for CTL that also supports fairness constraints. They claim that the fairness constraints can be arbitrary CTL formulae. The state enumeration version only allows fairness constraints to be state formulae (i.e. expressions that only depend on variables in current state with no temporal operators). Burch says these are equivalent [Bur93].

The arbitrary fairness constraints should give one the capability to reason about a pair of Estelle.Y modules that use an underlying reliable link as a communication channel. One should be able to specify temporal formulae that describe the behaviour of a reliable link as fairness constraints[1]. One also uses further fairness constraints to describe the behaviour of the users of the two modules that one wishes to assume takes place. For instance, one specifies that a sending user

[1]Using a temporal formula rather than a state formula for a fairness constraint is likely to seriously degrade the performance of the model checker.

infinitely often sends requests and that a receiving user infinitely often sends receive requests.

It is shown that using BDDs can significantly increase the size of systems that can be verified for certain useful classes of problems [BCMDH90]. Explicit state enumeration techniques are typically limited to systems with up to 10^6 states. Burch et al. verified a well-structured system with over 10^{20} states.

Here, binary decision diagrams and their operations are described. Then, the symbolic CTL model checking algorithm is described in detail.

3.2.1. Binary Decision Diagrams

A module that implements basic operators for manipulating Boolean functions in BDDs developed by Brace et al. [BRB90] is used in the CTL symbolic model checker. These operations include Boolean complement, conjunction and disjunction.

A BDD is a directed acyclic graph with all leaf nodes containing representations of either value zero or one, and all intermediate nodes containing a label for a Boolean variable and references to two other BDD nodes. Each intermediate node represents the Boolean function corresponding to the left node if the variable equals one and the Boolean function corresponding to the right node if the variables equals zero. For example, consider the graph in Figure 1 for an even parity function of four Boolean variables a,b,c,d.

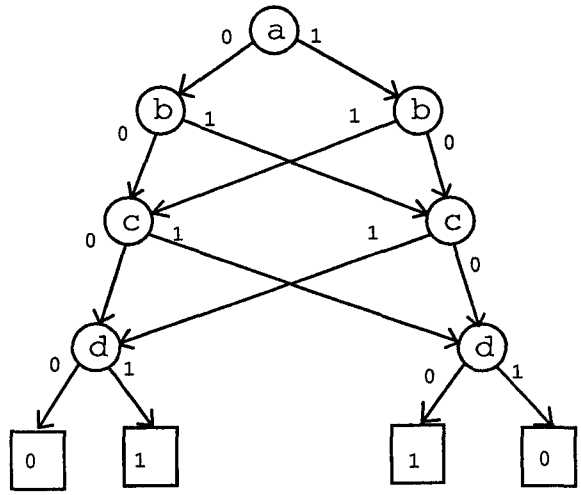


Figure 1. Binary Decision Diagram for Even Parity function of four Boolean variables

There are many other BDDs that could represent this same even parity function. Bryant introduced a restriction to the structure of BDDs to ensure each distinct function has a single canonical representation. This restriction is that the variables referenced in every path from top to bottom of the graph must be in the same order. The practical advantage of this restriction is that testing for equivalence of two functions simply involves testing whether the two graphs match exactly. Brace has optimized his implementation to the point that this operation is just a constant time pointer comparison.

As a consequence, one can test whether a function is satisfiable by just comparing the function's BDD with that of the zero function (i.e. False). In Brace's implementation, this operation takes constant time.

Brace's package supports a number of operations for manipulating a BDD to represent a database of Boolean functions. These include complement, conjunction, disjunction, composition of 2 functions (i.e. $\mathbf{f(g(}x\mathbf{))}$), testing for equivalence of 2 functions, and existential and universal quantification. The time complexity of the complement operation is proportional to the size of the function graph of the argument. The worst case time complexity of operators with 2 arguments is proportional to the product of the graph sizes of the 2 arguments.

The size of the BDD for a given function can be very sensitive to the chosen ordering of variables in the canonical BDD. Unfortunately, the problem of determining the best variable ordering for a given function is NP-complete (i.e. takes exponential time relative to the number of variables) [Bry86]. Malik et al. [MWBA88] consider heuristic strategies for determining the best ordering of variables for a given problem in an attempt to minimize the size of BDDs produced. Bryant says generally someone with a good understanding of the problem at hand can figure out a good variable ordering without much difficulty.

There are some functions such as multiplication that create an exponential sized BDD (relative to word size) no matter what variable ordering is chosen. Fortunately, in protocol specifications, arbitrary multiplication is not used too often.

Multiplication by a constant can be modelled as a series of bitwise shifts and additions.

3.2.2. Symbolic Model Checking Algorithm

Instead of modelling the current state of the implementation with the set S in M = (S,R,L), it is modelled with an array of Boolean variables $\vec{v}$. Suppose this array $\vec{v}$ has n elements. Then the number of states represented by $\vec{v}$ is 2^n.

Instead of associating an explicit label of a set of atomic proposition variables with each state $s \in S$ (namely L(s)), the state space $\vec{v}$ is defined by the implementation in such a way that the atomic propositions in a given state are derivable from the state $\vec{v}$ itself. Typically the implementation assigns an allocation of bits in $\vec{v}$ for control state information and the rest to logical variables in the source program. In fact, there are exactly n atomic propositions, one associated with each bit in $\vec{v}$. Thus the set of atomic propositions true in a given state s exactly corresponds to the 1 bits in the value s. All the other atomic propositions are false in this state.

Instead of representing the transition relation with $R \subseteq S \times S$, it is represented as a Boolean predicate of the current and next state values: $nextstate(\vec{v},\vec{v}')$. Logically this nextstate predicate says that there is a transition from v_1 to v_2 if and only if $nextstate(v_1,v_2) = True$.

Thus a model in the symbolic algorithm is defined as

$M = (\vec{v}, R)$ where $\vec{v}$ is an array of n Boolean variables

$R: \vec{v} \times \vec{v} \rightarrow Boolean$ is a predicate on current and next states.

Before explaining the symbolic model checking algorithm itself, the algorithm used by the *printtrace* command of the Ever verifier [HDDY92] is explained since ideas from it are used in the model checker's implementation.

Both the *printtrace* command and the model checker use the above described model of a system. Given a *start_state* predicate, a *next_state* predicate, and a *goal* predicate, *printtrace* does a reachability computation to determine whether the goal is reachable from any state satisfying *start_state*, and if so outputs a trace to it, otherwise outputs the largest trace attempting to reach the goal. *Start_state* and *goal* are predicates on only the current state, i.e. they are Boolean functions of the current state $\vec{v}$. *Next_state* is a Boolean function of the current and next states $\vec{v}$ and $\vec{v}'$. Starting from the *start_state* predicate, which logically represents the set of state values satisfying the start state condition, the *printtrace* algorithm computes the set of states that would be reached in the next time unit. This step is repeated until the goal state is reached (the intersection of the goal set and the last computed reached set is not empty) or a cycle back to the start set is detected (in which case the goal is unreachable from any state in the start set). This computation is illustrated in Figure 2.

```
compute_furthest_reachable_state(start,next,goal) ::=
  furthest_reached := start;
  while ((furthest_reached ∧ goal) = FALSE) do
    begin
    furthest_reached := furthest_reached ∨
                        ∃s'.((s=s') ∧
                             forward_image(furthest_reached(s),next(s,s')))
    { N.B. forward_image(furthest_reached(s),next(s,s')) ≡
           ∃s.furthest_reached(s) ∧ next(s,s') }
    if (furthest_reached = start) then break;  { cycle detected back to start }
    end;
  if ((furthest_reached ∧ goal) ≠ FALSE) then goal reached;
  if ((furthest_reached = start) then goal is unreachable;
                                 { i.e. goal is always false }
```

Figure 2 Computation of Furthest reachable state in Ever's printtrace command.

Logically forward_image(state_set, next) is just $\exists x.[state\_set \wedge next(x,x')]$. This is just the set of new states reachable from the original stateset *stateset* through the nextstate relation *next*. This is a Boolean function of a next state value saying whether that next state is reachable from one of the states in *stateset* through one iteration of the nextstate relation *next*. The first quantification operator in $\exists x'.((x = x') \wedge \exists x.[state\_set \wedge next(x,x')])$ is necessary to move the state values in the next phase to the current phase for the next time unit.

Once this algorithm has computed the furthest reached state, the same number of iterations of a reverse image computation are executed to print an example trace from the furthest reached state back to the start state. At each iteration, a particular state value is selected from the state set. This chosen state

value is printed out in terms of variables of the high-level Ever code specifying the nextstate relation.

Then a backward image is computed from this one example state value for the next iteration. The backward image computation is logically just $\exists x'.[stateset' \wedge next(x,x')]$ where stateset' is the set of states in the current time unit. This is a Boolean function of the current state value saying whether there is a transition in the nextstate that leads from this current state value to a state value in stateset'. In other words, it is the characteristic function of the set of states that lead to *stateset'* with one invocation of the nextstate relation *next*.

The above methods for computing forward and backward images are efficient provided the nextstate relation for the program specification has been fully evaluated as a BDD. For large problems, such as protocol specifications, it is not practical (in memory usage) to fully evaluate the nextstate relation [HDDY92]. Other researchers have proposed methods using Boolean Functional Vectors [CBM89,JPHS91,Fil91] that attempt to decompose the problem into a number of smaller problems whose results are later combined. Their systems only reason about deterministic systems. This is not sufficient for reasoning about the behaviour of communication protocols for modelling unpredictable events such as data loss on a noisy line. Fortunately, Hu et al. [HDDY92] show an efficient way of computing forward and backward images for non-deterministic

systems without building the BDD for the full nextstate relation. (They refer to this method as *image computation with disjunctive partitions*.)

Let us denote the nextstate relation with $N(x,x')$ where x is the current state and x' is the next state. They have derived the axioms in Tables 1 and 2 for forward and backward images, respectively, that reduce an image computation into the disjunction of a number of smaller images. X denotes a characteristic function of a set of states.

Type of Nextstate	**Forward Image Axiom**
composition	$forward\_image(X,(N_n \circ \ldots \circ N_1)(x,x')) =$ $forward\_image(forward\_image(X,N_1),$ $(N_n \circ \ldots \circ N_2)(x,x'))$
conditional	$forward\_image(X, if\ C(x)\ then\ N_1(x,x')\ else\ N_2(x,x')) =$ $forward\_image(X \wedge C, N_1) \vee forward\_image(X \wedge \overline{C}, N_2)$
disjunction	$forward\_image(X, N_1(x,x') \vee \ldots \vee N_n(x,x')) =$ $forward\_image(X,N_1) \vee \ldots \vee forward\_image(X,N_n)$
otherwise	$forward\_image(X,N(x,x')) = \exists x.[X(x) \wedge N(x,x'))$
*conjunction of disjunction	$forward\_image(X,(a_1 \vee \ldots \vee a_n) \wedge b) =$ $forward\_image(X,a_1 \wedge b) \vee \ldots \vee forward\_image(X,a_n \wedge b)$
*conjunction of conditional	$forward\_image(X,(\text{if } a \text{ then } b \text{ else } c) \wedge d) =$ $forward\_image(X \wedge a, b \wedge d) \vee forward\_image(X \wedge \neg a, c \wedge d)$

Table 1. Forward Image Axioms

Type of Nextstate	**Backward Image Axiom**
composition	$backward\_image(Y,(N_n \circ \ldots \circ N_1)(x,x')) =$ $backward\_image(backward\_image(Y,(N_n \circ \ldots \circ N_2)(x,x')),N_1)$
conditional	$backward\_image(Y,if\ C(x)\ then\ N_1\ else\ N_2) =$ $C \wedge backward\_image(Y,N_1) \vee$ $\overline{C} \wedge backward\_image(Y,N_2)$
disjunction	$backward\_image(Y,N_1 \vee \ldots \vee N_n) =$ $backward\_image(Y,N_1) \vee \ldots \vee backward\_image(Y,N_n)$
otherwise	$backward\_image(Y,N(x,x')) = \exists x'.[Y(x') \wedge N(x,x')]$
*conjunction of disjunction	$backward\_image(Y,(a_1 \vee \ldots \vee a_n) \wedge b) =$ $backward\_image(Y,a_1 \wedge b) \vee \ldots \vee backward\_image(Y,a_n \wedge b)$
*conjunction of conditional	$backward\_image(Y,(\text{if } a \text{ then } b \text{ else } c) \wedge d) =$ $(a \wedge backward\_image(Y,b \wedge d) \vee$ $(\neg a \wedge backward\_image(Y,c \wedge d)$

Table 2. Backward Image Axioms

All these axioms will reduce the size of BDDs constructed during the computation of the image usually with a higher requirement for CPU time.

An additional optimization for memory usage has been implemented for the computation of images. This is for the case of a nextstate relation that is a conjunction of disjunctions or a conjunction of conditionals. The corresponding additional axioms are marked with asterisks in Tables 1 and 2.

The actual CTL symbolic model checking algorithm which was added to the Ever verifier is now described.

To model check a given temporal formula f in the model associated with a given nextstate relation, the formula f is *evaluated* just like any type of unevaluated proposition. In the Ever verifier, an unevaluated proposition is a parse tree data representation of a formula associated with the proposition. Often when an unevaluated proposition is referenced in an expression which is being evaluated, the unevaluated proposition must be evaluated. When an unevaluated proposition has been evaluated, its value represented as a binary decision diagram is stored in memory with the unevaluated proposition's data structure. This BDD represents the characteristic function of the set of states in which the proposition just evaluated is true in the model associated with the given nextstate relation.

All the propositional logic operators are evaluated as before. No changes were made here. These operators include 3 basic operators (*and*, *or*, *not*), and high level operators that are essentially notational conveniences that are translated into the 3 basic operators (e.g. two integers being equal). These basic operators (i.e. *and*, *or*, *not*) are evaluated by invoking the primitives of Brace et al.'s BDD package [BRB90]

Three of the CTL temporal logic operators are evaluated as described below. The others are evaluated by using the duality properties of CTL to express them in terms of the three basic ones.

Let us denote the assumed nextstate relation of the model as $N(\vec{v}_i,\vec{v}_f)$. The unevaluated proposition *EXf*, which says there exists a path such that f is true in the next time unit, is evaluated as:

$\exists\vec{v}_f.[N(\vec{v}_i,\vec{v}_f)\wedge f(\vec{v}_f)]$

This expression is the same one as for a backward image. The EX operator is evaluated by computing the backward image of f with nextstate relation N.

The unevaluated proposition *E[f U g]*, which says there exists a path such that g is true sometime in the future and f is true in all preceding times along that path, is evaluated as the *least fixed point* of the expression $Z=g\vee(f\wedge EXZ)$. This formula is derived from the following idea. Let us assume that Z represents the proposition *E[f U g]*. At all times on the path that satisfies *f until g*, *g* must be true now, or *f* must be true now and *f until g* must be true in the next time unit. Our goal is to find a function *Z* which satisfies this property $Z=g\vee(f\wedge EXZ)$. Since this expression satisfies the monotone requirement of the Mu-calculus [BCMDH90], i.e. all free occurrences of Z fall under an even number of negations, the function *Z* that satisfies this property is the *least fixed point* of $g\vee(f\wedge EXZ)$. Since this expression is monotone, it means each time the assignment $Z:=g\vee(f\wedge EXZ)$ is re-evaluated (starting from *Z := False*), the size of the characteristic function increases. Thus eventually after some number of iterations, *Z* must reach a

fixpoint (i.e. its value remains the same for two successive iterations) or reach the point of being a tautology which is the largest possible set. Once this value of *Z* has been reached, *Z* is equal to $g \vee (f \wedge EXZ)$, so it satisfies the properties required by expression *E[f U g]*, so must be the evaluated value of *E[f U g]*. The algorithm for this computation is illustrated in Figure 3.

```
compute_eu(f,g) ::=
  Z1 := False;
  do
    Z := Z1;
    Z1 := g ∨ (f ∧ EXZ);
  until Z = Z1;
  return Z;
```

Figure 3. Computation of E[f U g] as least fixed point of Z=g∨(f∧EXZ)

An unevaluated proposition *EGf* is computed in a similar way to *E[f U g]* by using the fixpoint characterization formula $Z = f \wedge EXZ$. This is derived from the following property on a path making *EGf* true. At any state along such a path, *f* must be true now and *EGf* must be true in the following state (next time unit). Since this expression's primary operator is conjunction, it is a decreasing function, thus *EGf* is the greatest fixpoint of $Z = f \wedge EXf$ which is computed as illustrated in Figure 4.

```
compute_eg(f) ::=
  Z1 := True;
  do
    Z := Z1;
    Z1 := f ∧ EXZ;
  until Z = Z1;
  return Z;
```

Figure 4. Computation of EGf as greatest fixed point of Z=f∧EXf

Burch et al. [BCMDH90] present a technique for reducing the number of iterations needed for computing these fixpoints. It has not been implemented in this thesis. It is referred to as *iterative squaring*.

The other temporal operators are evaluated using the the duality properties of CTL in Figure 5.

$$EFf \equiv E[True\ U\ f]$$
$$AXf \equiv \neg EX\neg f$$
$$A[f\,U\,g] \equiv \neg E[\neg g\,U\,(\neg f \wedge \neg g)] \wedge \neg EG\neg g$$
$$AGf \equiv \neg EF\neg f$$
$$AFf \equiv \neg EG\neg f$$

Figure 5. CTL Duality Properties

3.2.3. Extension for fairness constraints

Burch et al. [BCMD90] devised an extension to the algorithm to modify it to consider only execution paths in the model satisfying a set of fairness constraints given by the user. They are specified as arbitrary CTL temporal formulae which are interpreted to be required to occur infinitely often on all considered paths. All execution paths where these constraints

do not occur infinitely often are ignored by the extended algorithm.

One should be able to specify desired assumptions of the environment's behaviour with these constraints. For example, with the alternating bit protocol, we want to verify its behaviour under the assumption that the users (sender and receiver) are infinitely often submitting requests to send and receive data, respectively.

The computation of the temporal operators are changed as follows. The formula EGf under fairness constraints $C_1,...,C_n$ means there exists a path from the current state in which f is globally true and all the constraints individually are true infinitely often along the path. This is characterized by the greatest fixed point of $Z = f \wedge EX(E[Z\,U(Z \wedge C_1)] \wedge ... \wedge E[Z\,U(Z \wedge C_n)])$. Note that each iteration in the fixpoint calculation involves the computation of n EU operators. These EU operators are presumably evaluated without fairness constraints, otherwise one would have endless recursion.

For a given model (i.e. nextstate relation), a value $Fair = EG\,True$ is calculated for use in computing the other temporal operators with fairness constraints. The computations for EX and EU are defined as follows:

$$EXf \equiv EX(f \wedge Fair)$$
$$E[f\ U\ g] \equiv E[f\ U(g \wedge Fair)]$$

All the other temporal operators can be computed by using the duality properties of CTL mentioned earlier (Figure 5).

3.3. Summary

The theories of CTL model checking were presented to provide a foundation for the next chapter. The data structures and algorithms of two main CTL model checking approaches (i.e. explicit and symbolic) were explained. The second approach is used in the implementation described in chapter 4. This second approach has great potential for reasoning about extremely large state machine systems.

Chapter 4: Implementation

In this chapter, the implementation of tools necessary for carrying out model checking verification of communication protocols is described. In section 4.1, the features of the Estelle.Y protocol specification language are explained. In section 4.2, the syntax and semantics of the Ever verifier's specification language are described. In section 4.3, the design of the translator that transforms Estelle.Y specifications into Ever code is given. The mappings from Estelle.Y language elements to Ever propositions is explained. In section 4.4, the way in which the symbolic model checking theories discussed in chapter 3 were applied to extend the Ever verifier to support CTL model checking is explained. In section 4.5, a number of minor enhancements made to Ever to make it more convenient for protocol verification are described.

In section 4.6, a CTL tableau construction algorithm that was implemented, is briefly described. Doing experiments with relatively trivial CTL formulae, it was concluded that this algorithm is too impractical for use in protocol verification.

In chapter 5, this implementation is applied to the alternating bit protocol.

4.1. Estelle.Y and ASN.1 Protocol Specification Language

Estelle.Y [Lu91] is a variant of the Estelle [ISO89] formal protocol description language. It is a language for describing a single module extended state machine containing a major state, internal variables, explicit timers and points of control and observation (PCOs).

The PCOs provide the module interfaces to the outside environment. Associated with each PCO is a queue in the module for holding interactions received from the environment awaiting servicing by this module. Zero or more input interactions (input service primitives (ISPs)) and zero or more output interactions (output service primitives (OSPs)) are associated with each PCO. These associations are defined in the ISP and OSP declaration sections of the Estelle.Y specification. Each interaction may contain a number of data fields. The names and types of these fields are defined in a corresponding type definition in the ASN.1 source file.

For example, the alternating bit protocol specification (see Appendices 1 and 2) has two points of control and observation: UAccessPoint (interface to user), NAccessPoint (interface to underlying link (network)). The interactions associated with these PCOs are illustrated in Table 3.

PCO	ISP
UAccessPoint	SENDrequest(x)
UAccessPoint	RECEIVErequest
NAccessPoint	DATAinteraction(id,conn,datum,seq)

PCO	OSP
UAccessPoint	SENDconfirm
UAccessPoint	RECEIVEresponse(x)
NAccessPoint	DATAinteraction(id,conn,datum,seq)

Table 3. List of interaction service primitives (ISPs) for points of control and observation (PCOs) in Alternating Bit Protocol Specification

The behaviour of an Estelle.Y module is defined by the set of transitions in the transition section. Each transition defines an action to be performed whenever a given condition is true. The action is specified with a Pascal-like compound statement that manipulates the data elements of the specification (namely, internal variables and timers) and some clauses shown in the output section of the table below. The condition is specified as a list of clauses whose types are given in Table 4.

Clause	Description
FROM state	true when module in given state
WHEN isp	true when an interaction of type isp is awaiting processing
PROVIDED boolean_expression	true when given expression is true
PRIORITY num	assigns this transition a priority
TO state	output clause stating new state after execution of transition
OUTPUT osp	states that transition output an interaction of type osp

Table 4. Estelle.Y transition clauses

The Pascal statements allowed in actions of Estelle.Y transitions are assignments, if statements, while statements and compound statements.

One or more timers are defined in the timer section of the specification. Each timer is assigned a name and a constant defining its upper count limit. Four additional statement types for timers are allowed in transition action code shown in Table 5.

Timer statement	**Description**
START(t)	put timer in running mode
STOP(t)	put timer in stopped mode
RESET(t)	reset timer's counter to zero and put it in stopped mode
SET(t,num)	set the timer's counter to given value

Table 5. Estelle.Y timer statements

Five types of timer expressions may be used in expressions of Pascal statements and in the PROVIDED clause of transitions, see Table 6.

Timer expressions	**Description**
TIMED_OUT(t)	timer counter reached limit defined in timer section
STARTED(t)	timer in running mode
STOPPED(t)	timer in stopped mode
RESETED(t)	timer in stopped mode with counter equal to zero
READ(t)	returns value of timer's counter

Table 6. Estelle.Y timer expressions

The data type of each internal variable defined in the variable declaration section must be Boolean, Integer or character string.

A complete specification of the syntax of the Estelle.Y language is given in [Lu91].

Note it is not necessary for an Estelle.Y specification to be written in such a way as to have exactly one transition enabled at a time (i.e. does not have to be deterministic). If more than one transition is enabled, the specification says that the implementation must choose one of them. If none are enabled, this module must remain in its current state (i.e. major state and internal variables) until one becomes enabled.

4.2. Ever specification language

Ever is a language for specifying deterministic and non-deterministic state machines with high-level language type constructs. It is also a tool for doing reachability analysis of systems specified in its language. The syntax of types, variables, bit vectors, propositions and predicates is described. Then Ever's main function called *printtrace* for displaying a trace after doing reachability analysis is explained.

First, it provides commands for defining the data types and variables used in the specification. All types ultimately

reduce to sequences of bits. The types are sequences of bits, records and arrays. A new named type is defined with the *deftype* command:

deftype *typename typedefinition*;

Valid types are illustrated in Table 7.

Type	**Description**
(bits *n*)	sequence of *n* bits containing values 0 through $2^n - 1$
(array *l u type*)	array indexed from lower bound *l* to upper bound *u* containing elements of type *type*
(record *f1 t1 ... fn tn*)	structure containing fields named *f1* through *fn* of types *t1* through *tn*, respectively

Table 7. Ever data types

Variables are defined with the *defvar* command as follows:

defvar *varname type*;

where *varname* is name assigned to variable
type is named type or an actual type definition

The values of all variables are bit vectors. There are some bit vector operators for manipulating values. Thus bit vectors are numerical constants, references to variables, and various operators on sequences of bits such as arithmetic operations. Some are shown in Table 8.

Bit vector	**Description**
c	numerical constant *c* represented in as few bits as possible
'size'c	numerical constant *c* represented in *size* bits
v^c	reference to variable *v* in current phase which can be a reference to an element of an array or a field of a record
v^n	reference to variable *v* in next phase which can be a reference to an element of an array or a field of a record
v^p	reference to variable *v* in previous phase which can be a reference to an element of an array or a field of a record
(add *v1 ... vn*)	the sum of the bit vectors *v1* through *vn*
(sub *v1 v2*)	the difference between bit vectors *v1* and *v2*

Table 8. Ever bit vector types

Third, Ever provides commands for defining propositions that may depend on the defined variables in the current and next phases. (Ever also provides syntax for dealing with variables in the previous phase but the reachability analysis algorithm is not designed to use it properly.) The main types of propositions are shown in Table 9.

Proposition	**Description**
TRUE	returns true
FALSE	returns false
CurNextEq	returns true when each variable has same value in current and next phases.
(and *p1 ... pn*)	returns true when all propositions *p1* through *pn* are true
(or *p1 ... pn*)	returns true when at least one of the propositions *p1* through *pn* is true
(not *p*)	return true when proposition *p* is false
(implies *a c*)	return true when antecedent proposition *a* is false or consequent proposition *c* is true
(equiv *p1 p2*)	returns true when propositions *p1* and *p2* are identical
(eq *bv1 bv2*)	returns true when bit vector *bv1* is equal to bit vector *bv2*
(gt *bv1 bv2*)	returns true when bit vector *bv1* is greater than bit vector *bv2*
(ge *bv1 bv2*)	returns true when bit vector *bv1* is greater or equal to bit vector *bv2*
(lt *bv1 bv2*)	returns true when bit vector *bv1* is less than bit vector *bv2*
(le *bv1 bv2*)	returns true when bit vector *bv1* is less than or equal to bit vector *bv2*
(if *c p1*)	returns true if *c* is false and returns true if *c* is true and *p1* is true[1]
(if *c p1 p2*)	returns true if *c* is true and *p1* is true or if *c* is false and *p2* is true.
(becomes *v*^n *bv*)	returns true if all variables except *v* have same value in current and next phases and variable *v* has value of bit vector *bv* in the next phase.[2]
(compose p1 ... pn)	returns true if when the values of the variables in the current and next phases are related in a way equivalent to the effect of applying the relations (of current and next phases) p1 through pn in succession.[3]

Table 9. Ever propositions

[1]The conditional proposition if assumes the condition c is a proposition only on variables in the current phase.

[2]The *becomes* proposition assumes that the variable v is in the next phase. v may be reference to an array element or a field of a record.

[3]The *compose* proposition assumes that the propositions *p1* through *pn* are relations between all variables in current and next phases (i.e. *becomes*, *compose*, conditionals with propositions relating current and next phases in *then* and *else* parts) propositions.

Ever supports a notion of predicates which are just a syntactic convenience to avoid having to rewrite propositions with same structure but different constants substituted in places. It is a parameterized proposition whose actual definition depends on the set of values passed to the predicate when it is instantiated. The syntax is as follows:

defpred *predicate_name* (*p1* ... *pn*) *proposition*;

where *p1* through *pn* are named constants whose values are determined when the predicate is instantiated

proposition is a proposition whose definition may reference named constants *p1* through *pn*

Ever's main function, *printtrace*, given a *nextstate* proposition attempts to find a execution trace from any state that satisfies the given *start* proposition to any state that satisfies the given *goal* proposition. The nextstate proposition is assumed to relate the current and next phases of the model in which a trace is being searched. In other words, the nextstate proposition evaluates to true when given a pair of current and next phase variable value sets that can occur in the corresponding model (i.e. when the next phase variable values set can be reached from the current variable values set in one time step). If such a trace is found, it is printed as a set of current variable values for each time step from one satisfying the *start* proposition to one satisfying the *goal* proposition. If no such trace is found, the longest one tried is printed.

The theory upon which this algorithm is based was described in the previous chapter.

4.3. Estelle.Y to Ever translator

In this section, the translation from Estelle.Y to Ever is described. The procedure for the user to carry out this translation is described. The way a set of Estelle.Y modules is represented in Ever is explained. Then a description of the algorithm is given.

Given a list of pairs of source files (Estelle.Y file and ASN.1 binary file) for the types of modules to be included, *estelle2ever* produces corresponding Ever code in the standard output. For example to produce code for two alternating bit protocol modules directly tied together, the user invokes *estelle2ever* as follows. Note the user's responses are shown in italics. See the Estelle.Y source code of the alternating bit protocol in Appendix 1.

```
1% estelle2ever abp5.estelle abp5.tt > abp5.output
Number of instances of module 'abp5' : 2
connect abp5[0].UAccessPoint to :
connect abp5[0].NAccessPoint to : abp5[1].NAccessPoint
connect abp5[1].UAccessPoint to :
2%
```

To generate code for a system without the two modules' NAccessPoint interaction points directly tied together the session would be:

```
1% estelle2ever abp5.estelle abp5.tt > abp5.output
Number of instances of module 'abp5' : 2
connect abp5[0].UAccessPoint to :
connect abp5[0].NAccessPoint to :
connect abp5[1].UAccessPoint to :
2%
```

The representation of a set of Estelle.Y module instances in Ever code is described. (For each Estelle.Y source file given on the command line, types are defined for its internal variables, timers, major state and points of control and observation (PCOs).) An array variable is defined for each Estelle.Y source file to hold all the internal variables, major state and timers of all instances of the corresponding module. For the alternating bit protocol these definitions are as follows:

```
-- Variables:
deftype abp4_localvars (record
    Send_seq (bits 1)
    Recv_seq (bits 1)
    Recv_buffer_empty (bits 1)
    Recv_buffer_datum (bits 1)
    Recv_buffer_seq (bits 1)
    Send_buffer_empty (bits 1)
    Send_buffer_datum (bits 1)
    Send_buffer_seq (bits 1));

-- Timers: rexmit_timer
deftype abp4_timers_type (record
    rexmit_timer (record running (bits 1) counter (bits 2)));

-- States: ack_wait estab
deftype abp4_mainstate (bits 1 "ack_wait" "estab");

-- Ever variables for module abp4 states
defvar abp4 (array 0 1 (record state abp4_mainstate vars abp4_localvars
                               timers abp4_timers_type));
```

An array variable is defined for each role of each PCO type encountered in all the modules in the specification. The PCOs need to be stored separately from the modules' other variables because they need to be accessed by code of more than one module type. The alternating bit protocol has two PCOs: UAccessPoint and NAccessPoint. The two roles of the UAccessPoint PCO are different and the roles of the NAccessPoint PCO are the same. The NAccessPoint roles were defined in a way to be the same to allow one to directly connect a pair of Estelle.Y alternating bit protocol modules. The translator only allows matching roles to be connected.

```
-- ISPs: SENDrequest RECEIVErequest DATAinteraction

-- OSPs: RECEIVEresponse DATAinteraction SENDconfirm

-- PDUs: Junk

-- PCOs:
deftype UAccessPoint_inqueue (record
    kind (bits 1)
    RECEIVErequest  (record dummy (bits 1))
    SENDrequest  (record udata (bits 1)));

deftype UAccessPoint_outqueue (record
    kind (bits 1)
    SENDconfirm  (record dummy (bits 1))
    RECEIVEresponse  (record udata (bits 1)));

deftype NAccessPoint_queue (record
    DATAinteraction  (record ndata (record id (bits 1) conn (bits 1)
                              data (bits 1) seq (bits 1))));

-- Ever variables for interaction queues
defvar UAccessPoint_in (array 0 1
                          (record base (bits 0) queued (bits 1)
                                  content (array 0 0 UAccessPoint_inqueue)));
defvar UAccessPoint_out (array 0 1
                          (record base (bits 0) queued (bits 1)
                                  content (array 0 0 UAccessPoint_outqueue)));
defvar NAccessPoint (array 0 1
                          (record base (bits 0) queued (bits 1)
                                  content (array 0 0 NAccessPoint_queue)));
```

A special variable is defined for the implementation of timers. This variable is special because it is defined to be *free* meaning that it is excluded from the set of variables that must remain stable in a *becomes* proposition. In other words this variable's current value is totally non-deterministic. Its value is used in the nextstate relation to decide whether to treat timers or transitions in the current time unit. During verification, two fairness constraints are defined to ensure this variable is infinitely often true and is infinitely often

false. These constraints are used to make the verification consider only execution paths where both transitions and timers are regularly executing.

```
deffreevar clock_tick (bits 1);
```

For each type of Estelle.Y module in the specification, a pair of Ever predicates are defined for each transition in the module. The first predicate defines the enabling condition of the transition. The parameters of the predicate are a module instance number and indices to all input and output service primitives arrays used by the module. It is a conjunction proposition of propositions translated from the clauses of the transitions as shown in Table 10.

Clause	**Propositions**
FROM *state*	(eq *modulename*[n].state^c *state_code*)
WHEN *isp*	(not (eq *pco_role*[i].queued^c 0)) (eq *pco_role*[1].content[*pco_role*[i].base^c].kind^c *isp_code*)
PROVIDED *boolean_expression*	Ever code for *boolean_expression*
PRIORITY *num*	(not implemented)
OUTPUT osp	(lt *pco_role[o]*.queued^c QSIZE)[4]

Table 10. Ever propositions for Estelle.Y transition clauses

The second predicate defines the relationship between the state before and after the action code of the transition is executed. Its parameters are a module instance number, and

[4]Note QSIZE is a constant defining the number of elements in all interaction queues. This condition blocks a transition that would normally add an interaction to a queue from executing when the queue is already full.

indices to used ISP arrays and OSP arrays. It is a *compose* proposition containing a proposition for each Pascal-like statement in the transition action code. The statements are translated as indicated in Table 11.

Statement	**Proposition**
v := *expr*	(becomes *v*^n *expr*)
if *c* then *s1*	(if *c s1* (becomes *modname*^n *modname*^c))[5]
if *c* then *s1* else *s2*	(if *c s1 s2*)
begin *s1* ... *sn* end;	(compose *s1* ... *sn*)
while c s1	not implemented
START(*t*)	(becomes *modname*[n].timers.*t*.running^n 1)
STOP(*t*)	(becomes *modname*[n].timers.*t*.running^n 0)
RESET(*t*)	(becomes *modname*[n].timers.*t*^n 0)
SET(*t*,*num*)	(becomes *modname*[n].timers.*t*.counter^n *num*)

Table 11. Ever propositions for Estelle.Y statements

The destination variable in the assignment statement is translated as the following if *v* is an internal variable of the module:

modname[n].vars.*v*

It is translated as the following if the destination variable is in an output service primitive:

pco_role[o].content[(add *pco_role*[o].base^c *pco_role*[o].queued^c)].*osp_name*

Additional actions must be performed for transitions with the following clauses to complete the transaction of executing the transition (see Table 12).

[5]Note the binary *if* needs the dummy *becomes* proposition in its *else* part to state that all variables must remain the same in that case. Otherwise, the TRUE proposition would be placed in the *else* part which would mean that anything can happen when the condition of the if statement is false.

Clause	Additional Actions
TO *state*	(becomes *modname*[n].state^n *state_code*)
WHEN *isp*	(becomes *pco_role*[i].base^n (add *pco_role*[i].base^c 1)) (becomes *pco_role*[i].queued^n (sub *pco_role*[i].queued^c 1))
OUTPUT *osp*	(becomes *pco_role*[o].content[(add *pco_role*[o].base^c *pco_role*[o].queued^c)].kind^n *osp_code*) (becomes *pco_role*[i].queued^n (add *pco_role*[i].queued^c 1))

Table 12. Ever propositions for actions for completing Estelle.Y transitions

Next a predicate is defined for handling counting of timers. For each timer, the timer's counter is incremented if it in the running mode and the maximum count has not been reached. If none of the timers are running, this predicate is a dummy *becomes* proposition to keep all (non-free) variables stable. For the alternating bit protocol, it is as follows:

```
defpred abp6_timertick (n) (compose
    (if (and (eq abp6[n].timers.rexmit_timer.running^c 1)
             (lt abp6[n].timers.rexmit_timer.counter^c 3))
        (becomes abp6[n].timers.rexmit_timer.counter^n
                 (add abp6[n].timers.rexmit_timer.counter^c 1))
        (becomes abp6[n].timers^n abp6[n].timers^c)));
```

A predicate defining the overall behaviour of a module instance is defined. It is a disjunction of all possible actions that the module may perform at each time unit: service counting of timers if variable *clock_tick* is true, and service each enabled transition if variable *clock_tick* is false. It is a disjunction because we want the specification to be non-deterministic allowing any enabled transition to be serviced. This disjunction includes a case to keep all variables stable

when all transitions are disabled. If this special case were not included, the whole proposition would evaluate to a fallacy leading to a deadlock when all transitions are disabled. This predicate in the alternating bit protocol specification is as follows. (Note each transition is defined with an *if* proposition with a false *else* part. This is done to persuade the backward image computing algorithm discussed in the next section to use its partitioned nextstate evaluation technique.)

```
defpred abp6_nextstate (n i0 o0 i1 o1) (or
    (if (eq clock_tick^c 0) (or
        (if (abp6_Ctrans1 n i0 o0 i1 o1) (abp6_Atrans1 n i0 o0 i1 o1) FALSE)
        (if (abp6_Ctrans2 n i0 o0 i1 o1) (abp6_Atrans2 n i0 o0 i1 o1) FALSE)
        (if (abp6_Ctrans3 n i0 o0 i1 o1) (abp6_Atrans3 n i0 o0 i1 o1) FALSE)
        (if (abp6_Ctrans4 n i0 o0 i1 o1) (abp6_Atrans4 n i0 o0 i1 o1) FALSE)
        (if (abp6_Ctrans5 n i0 o0 i1 o1) (abp6_Atrans5 n i0 o0 i1 o1) FALSE)
        (if (abp6_Ctrans6 n i0 o0 i1 o1) (abp6_Atrans6 n i0 o0 i1 o1) FALSE)
        (if (abp6_Ctrans7 n i0 o0 i1 o1) (abp6_Atrans7 n i0 o0 i1 o1) FALSE)
        (if (and
                (not (abp6_Ctrans1 n i0 o0 i1 o1))
                (not (abp6_Ctrans2 n i0 o0 i1 o1))
                (not (abp6_Ctrans3 n i0 o0 i1 o1))
                (not (abp6_Ctrans4 n i0 o0 i1 o1))
                (not (abp6_Ctrans5 n i0 o0 i1 o1))
                (not (abp6_Ctrans6 n i0 o0 i1 o1))
                (not (abp6_Ctrans7 n i0 o0 i1 o1)))
            (becomes abp6^n abp6^c) FALSE)) FALSE)
    (if (eq clock_tick^c 1) (abp6_timertick n) FALSE));
```

Finally, the overall nextstate relation for all instances of all modules is defined that invokes the modules' nextstate predicate with different parameters for each module instance. The parameters specify the index of each PCO array to be used by each module instance.

```
defprop nextstate (or
    (abp4_nextstate 0 0 0 0 1)
    (abp4_nextstate 1 1 1 1 0));
```

A predicate is defined for each module type to define the initial state of that module. For example,

```
defpred abp4_init (n o0 o1) (and
    (eq abp4[n].state^c 1)
    (eq NAccessPoint[o1].base^c 0)
    (eq NAccessPoint[o1].queued^c 0)
    (eq UAccessPoint_out[o0].base^c 0)
    (eq UAccessPoint_out[o0].queued^c 0)
    (eq abp4[n].vars.Send_seq^c 0)
    (eq abp4[n].vars.Recv_seq^c 0)
    (eq abp4[n].vars.Send_buffer_empty^c 1)
    (eq abp4[n].vars.Recv_buffer_empty^c 1)
    (eq abp4[n].timers^c 0));
```

Then a proposition for the overall initial state of all modules is defined. For example,

```
defprop init (and
    (abp4_init 0 0 0)
    (abp4_init 1 1 1));
```

The algorithm for generating Ever code from a set of Estelle.Y specifications is now described. Each of the specifications is parsed with a slightly modified copy of the *pdsparse()* routine from TESTGEN (a test sequence generation package for Estelle.Y) into a protocol data structure (PDS) [Lu91,VHLMD93].

All the PDS structures are scanned to build a list of all PCOs in all modules. This list is scanned to query the user for information about number of instances of each Estelle.Y module and which pairs of PCOs are to be connected. Type and variable declarations for all the PCOs are produced. A module instance data structure is created for each instance of each Estelle.Y

module. It includes a module instance number, index numbers to elements of PCO arrays from which this module instance shall receive interactions, and index numbers to elements of PCO arrays to which this module instance shall send interactions. These module instance structures are not used until the global nextstate relation and initial state propositions are produced.

For each module type, Ever predicates (as described above) for the transitions conditions and actions are produced from information in the PDS.

The global nextstate and initial state propositions are generated from the module instance data structures.

While statements are not supported because loop analysis techniques of symbolic evaluation would have to be used to derive expressions for the overall effect of loops. Ever does not support such a construct.

If a while loop in the action code of an Estelle.Y transition were to be translated to a series of transitions handling each iteration of the loop, a Boolean variable would be needed to indicate that the loop is executing. This would be necessary to indicate that no input events can be processed while executing the loop. The action code of an Estelle.Y transition is supposed to be atomic.

Prioritized transitions were not implemented. The algorithm could be enhanced to generate a more sophisticated expression for deciding which transitions are enabled.

Arbitrary integer multiplication and division is not supported because these operations are not supported in Ever. BDDs are known to blow up badly with these operations.

It would be useful if the Estelle.Y language allowed one to specify an explicit data size when declaring (internal) integer variables. For example, the LAPB specification uses 3-bit sequence numbers. In the alternating bit protocol specification, Boolean variable declarations were used for the 1-bit sequence number variables to avoid this problem.

4.4. Ever extensions for CTL Model Checking

This section describes procedures added to the Ever verifier to support the evaluation of CTL temporal logic formulae. These procedures were added to the *evaluate_proposition()* routine where all types of expressions are evaluated. Non-temporal expressions are directly evaluated by invoking primitives of Brace's BDD package. The EX, EU and EG operators are computed as described below. The other five operators (EF, AX, AU, AG, AF) are computed by expressing them in terms of the first three using duality properties of the CTL logic.

The expression *EXf* is evaluated by doing some manipulations to move from the current phase of variables to the next phase of variables and computing the backward image of the characteristic set of *f*. The routine is illustrated below:

```
bdd_ex(f,next) ::=
{ following line produces a characteristic function for f of
  variables in next phase instead of in current phase as
  when passed to this function }
temp1 := ∃cur_vars.((next_vars = cur_vars) ∧ f(cur_vars);

{ determine set of states from which all possible transitions
  will reach set of states defined by temp1 }
{ temp1 is a characteristic function of variables in the next phase.
  Compute_backward_image returns a characteristic function of
  variables in current phase. }
return compute_backward_image(temp1, next);
```

The routine for computing backward images for Ever's *printtrace* command is used. This routine uses a partitioning technique to avoid evaluating the complete nextstate relation whenever possible. This technique dramatically reduces the amount of memory required. The axioms used for this partitioning are as follows (recall Table 2 on page 40):

```
{ backward image of a composition }
backward_image(Y, (Nn°...°N1)(x,x')) ≡
   backward_image(backward_image(Y, (Nn°...°N2)(x,x')), N1)

{ backward image of a condition (i.e. if then else statement) }
backward_image(Y, if C(x) then N1 else N2) ≡
   C ∧ backward_image(Y, N1)  ∨  ¬C ∧ backward_image(Y, N2)

{ backward image of a disjunction }
backward_image(Y, N1 ∨ ... ∨ Nn) ≡
   backward_image(Y, N1) ∨ ... ∨ backward_image(Y, Nn)

{ backward image is generally defined as }
backward_image(Y, N(x,x')) ≡ ∃x'.(Y(x') ∧ N(x,x'))
```

EU expressions are evaluated by performing a least fixpoint computation by repeatedly using the EX computation procedure until a fixed value is reached:

```
bdd_eu(f,g,next) ::=
  Z1 := False;
  do
    Z := Z1;
    Z1 := g ∨ (f ∧ EXZ);
  until Z = Z1;
```

EG expressions are evaluated by performing a fixpoint computation by repeatedly using the EX procedure until a fixed value is reached:

```
bdd_eg(f,next) ::=
  Z1 := True;
  do
    Z := Z1;
    Z1 := f ∧ EXZ
  until Z = Z1
```

The other CTL operators are evaluated by using the duality properties to express them in terms of the three operators above:

```
EFf = E[True U g]
AXf = ¬EX¬f
A[f U g] = ¬E[¬g U (¬f ∧ ¬g)] ∧ ¬EG¬g
AGf = ¬EF¬f
AFf  = ¬EG¬f
```

Further enhancements were made to these routines to support fairness constraints. The computation of the EG operator was modified to replace `Z := f ∧ EXZ` with the following where C1 through Cn are the fairness constraints:

```
Z1 := f ∧ EX(E[Z U (Z ∧ C1)] ∧ ... ∧ EX(E[Z U (Z ∧ Cn)])
```

Note the EU expressions within this assignment statement are evaluated without using the fairness constraints. This would result in an infinite loop computing fairness constraints.

The EX and EU operators are evaluated as follows under fairness constraints where Fair = EG True. Fair is evaluated with fairness constraints and the two expressions below for the EX and EU operators are evaluated using the procedure without fairness constraints.

```
EXf :       EX(f ∧ Fair)
E[f U g] : E[f U (g ∧ Fair)]
```

4.5. Improvement to Ever printing of propositions

The original version of Ever printed propositions without any symbolic names for referenced variables. An improvement was made to have meaningful names printed. This improvement greatly eases reading and interpreting results from the verifier. Figures 6 and 7 illustrate the difference between the old and new versions.[6]

[6]The *printsop* command prints a BDD result as a sum of products (disjunction of conjunctions).

```
1% ever
Limited memory recycling for debugging is ON
  memory cycle length: 10000
  added overhead: 160000 bytes
  set/reset by defining BDD_MEMORY_DEBUG in bdd.h to 1/0
Command> defvar p (bits 1);
Command> defvar x (bits 3);
Command> printprop (eq p^c 1);
(:301024:2:1)
Command> printsop (eq p^c 1);
(:301024:2:1)
Command> printprop (eq x^c 6);
!(:302384:5:4
   [1]
   !(:303936:8:7
      (:301504:11:10)
      [0]))
Command> printsop (eq x^c 6);
!(:302384:5:4
   [1]
   !(:303936:8:7
      (:301504:11:10)
      [0]))
Command> end;
2%
```

Figure 6. Demonstration of Ever without proposition printing improvement

```
1% ever
Limited memory recycling for debugging is ON
  memory cycle length: 10000
  added overhead: 160000 bytes
  set/reset by defining BDD_MEMORY_DEBUG in bdd.h to 1/0
Command> defvar p (bits 1);
Command> defvar x (bits 3);
Command> printprop (eq p^c 1);
(p^c)
Command> printsop (eq p^c 1);
p^c +
Command> printprop (eq x^c 6);
!(x.bit0^c
   [1]
   !(x.bit1^c
      (x.bit2^c)
      [0]))
Command> printsop (eq x^c 6);
!x.bit0^c & x.bit1^c & x.bit2^c +
Command> end;
2%
```

Figure 7. Demonstration of Ever with proposition printing improvement

Brace's BDD package [BRB90] provides a capability to store user defined information with each bit variable. A data structure was defined to refer back to the original definition of the variable in a way so that a meaningful name can be printed for all variables references including array elements and record fields.

4.6. Addition of Ever *deffreevar* command

The *deffreevar* command allows one to define a variable which will not be assumed to have its value remain stable from one time unit to the next when no other assumption is specified.

The *becomes* Ever proposition operator normally generates a proposition saying that the specified destination variable shall hold the value of the given expression in the next time unit and that all other variables shall remain stable. The *deffreevar* command excludes its variables from the set of variables to remain stable in *becomes* propositions.

This command was necessary to model a non-deterministic (i.e. not predetermined by model) sequence of input data being sent by a sending module when verifying the alternating bit protocol. In other words, this command allowed the verification of the alternating bit protocol to reason about all possible input data sequences.

Figure 8 illustrates the effect of the *deffreevar* command. Note, the *printsop* command prints a proposition as a sum of products. The *becomes* expressions for assigning the value 0 to variable *p* does not require the variable *q* to remain stable because *q* is defined to be free.

```
1% ever
Limited memory recycling for debugging is ON
  memory cycle length: 10000
  added overhead: 160000 bytes
  set/reset by defining BDD_MEMORY_DEBUG in bdd.h to 1/0
Command> defvar p (bits 1);
Command> deffreevar q (bits 1);
Command> printsop (becomes p^n 0);
!p^n +
Command> printsop (becomes q^n 0);
p^c & p^n & !q^n +
!p^c & !p^n & !q^n +
Command> end;
2%
```

Figure 8. Demonstration of new deffreevar Ever command

4.7. Addition of *setdefaultnextstate* command

All the CTL temporal operators in Ever when originally implemented required a nextstate relation to be explicitly specified. This command was added to provide a notational convenience for the user. With this new command, when no nextstate relation is specified with a CTL operator, the default is assumed.

The implementation of the fairness constraints support was done in such a way as to only work correctly when the default nextstate relation is used. The value of *Fair=EGTrue* is stored with the value of the default nextstate relation. *Fair=EGTrue* is evaluated when the first temporal expression requiring its value is evaluated.

4.8. Temporal logic to Nextstate relation translation algorithm

A strategy for carrying out incremental verification was attempted that combines model checking and tableau construction techniques. From a few simple experiments, we concluded that this approach is infeasible because the complexity of the tableau construction algorithm is exponential and formulae to express simple properties are surprisingly large. The strategy and algorithms are briefly described below.

The tableau construction algorithm [CE81] is used to translate a CTL formula describing the behaviour of the

environment surrounding the Estelle.Y modules (being verified) into a tableau representation. Clarke and Emerson describe a method [CE81] to derive a model from the constructed tableau. However, we need to produce a non-deterministic finite state machine representing all possible models satisfying the property defined by the input CTL temporal logic formula. The tableau is translated into a non-deterministic finite state machine with an extension made to the tableau algorithm. This machine is transformed into a nextstate relation.

The Estelle.Y modules are translated into Ever code with the *estelle2ever* translator. A non-deterministic finite state machine for all the Estelle.Y modules is produced.

A global nextstate relation is defined as the conjunction of all the nextstate relations for the environment behaviour and the Estelle.Y modules' behaviour. Conjunction is used to model the parallel execution of all the components.

Disjunction could not be used because it would include invalid behaviours. The tableau produces a graph representing all possible behaviours of a given formula. The problem is that the tableau has no notion of input variables and includes behaviours that are contradictory with other modules' outputs. A conjunction of the nextstate relations of all the components cancels out contradictory behaviours. If a disjunction could be used, the model checking would use far less memory because the

partitioned nextstate evaluation techniques in Ever could be used. See section 4.4 above.

This global nextstate relation is then passed to the model checker to check properties of the global system.

Clarke and Emerson's tableau construction algorithm for CTL formulae [CE81] was implemented to experiment with this strategy.

A new algorithm was developed to produce a non-deterministic state machine representing all possible behaviours of the input CTL formula from the tableau constructed by the tableau algorithm.

4.9. Summary

Before presenting the Estelle.Y to Ever translator algorithm, the syntax and semantics of the Estelle.Y and Ever languages were reviewed. Then the conventions about how a number of Estelle.Y modules are represented in the Ever language were explained before describing the translation algorithm itself.

The Ever verifier was extended to support CTL model checking with fairness constraints. In the process of integrating the symbolic model checking algorithms, discussed in chapter 3, into the verifier, a few new commands and features were added to support the model checking.

Chapter 5: Experiments and results

In this chapter the tools developed in the previous chapters are applied to the alternating bit protocol. Section 5.1. explains the strategy used to verify the control structure of the protocol. It proposes to prove the correctness of the protocol by showing that a number of internal properties are true and giving an argument as to why the protocol is correct when these internal properties are true. Further properties are specified to show that the protocol is valid for all data packet sizes.

Section 5.2 discusses the results of the verification. The main result is that all the formulae verified to be tautologies and that they evaluated in finite time and memory. A method for producing counterexample traces is proposed and demonstrated with an example from the alternating bit protocol.

5.1. Verifying the Alternating Bit Protocol

The alternating bit protocol is a classical protocol for providing a reliable flow of messages from a sending process to a receiving process over an unreliable channel that may delay, (repeat) and lose messages (but not reorder the sequence of messages on the channel).

Its Estelle [ISO89] specification was manually translated into Estelle.Y and ASN.1 (listed in Appendices 1 and 2). The modular decomposition of this specification is illustrated in Figure 9.

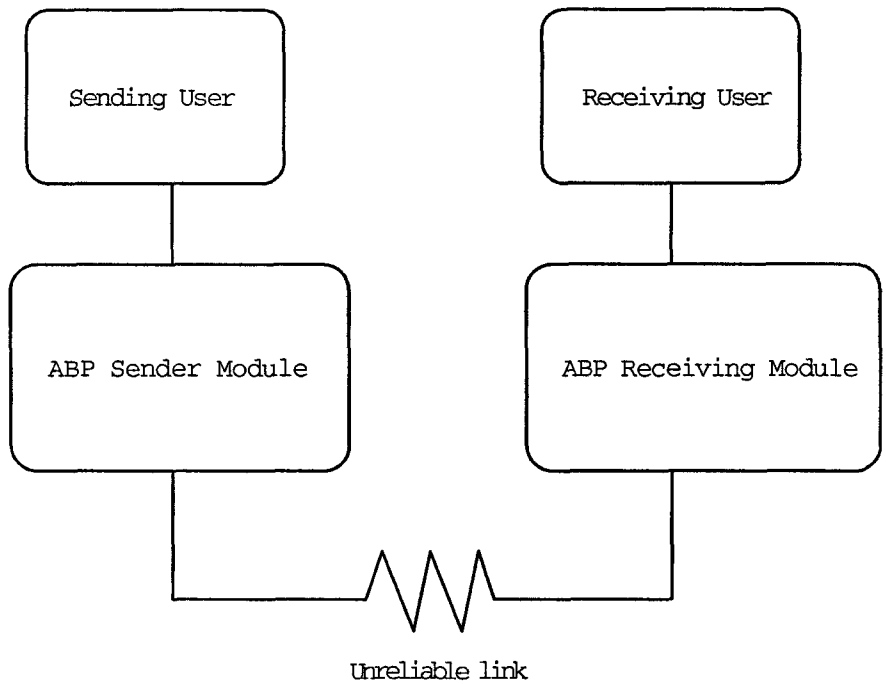


Figure 9. Modular Decomposition of Alternating Bit Protocol Specification

The following assumptions were made during the translation from Estelle. The data contained in each message is a single bit. This choice was made to minimize the size of the BDD produced during verification. We argue that this data size is sufficient for proving properties of the control structure of the protocol. We assume that the underlying unreliable link drops packets that are corrupted by noise on the channel.

The interaction queues of the Estelle.Y modules only have sufficient capacity for holding one request at a time. Under the assumption that the users of the protocol module obey an *interface protocol* (i.e. never fail to respond to interactions delivered to the interfaces in an appropriate manner), we show that the ABP protocol provides a reliable service.

It is not sufficient to give the model checker only a specification of the protocol modules behaviour. Since the model checker needs a description of a whole system, we must add information about the behaviour of the user modules and the underlying data link.

We specified all the possible behaviours of the user modules in the simplest way possible to minimize the complexity of the system given to the model checker. For the sending user, we specify two possible actions:

1. Whenever the send queue to ABP sending module is empty, the user may add a random data send request to the queue.
2. Whenever a SendConfirm interaction is in the queue from the ABP sending module, the sending user may dequeue this interaction.

For the receiving user, we specify two possible actions:

1. Whenever the request queue to the receiving ABP module is empty, the receiving user may submit a ReceiveRequest interaction to the queue.

2. Whenever a ReceiveResponse interaction is in the queue from the receiving ABP module, the receiving user may dequeue this interaction (i.e. receive the data originally sent by the sending user).

We model the behaviour of the underlying data link by just tying the lower interaction points of the ABP modules directly together when translating the Estelle.Y specification into Ever code. This is shown in Figure 10. This is sufficient for modelling a system with a reliable underlying link.

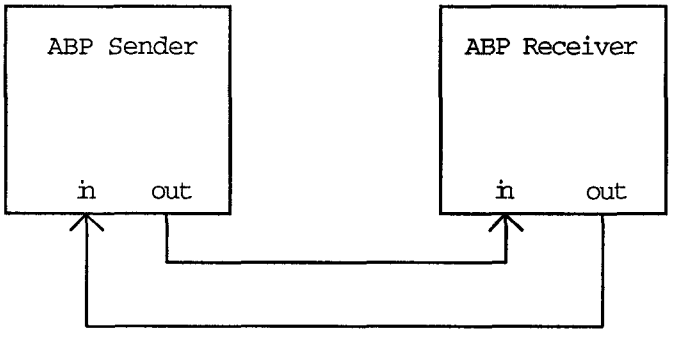


Figure 10. Alternating Bit Protocol module configuration for modelling reliable underlying link and modelling unreliable link that can only lose data packets

For an unreliable link that only may lose data but not duplicate data or reorder data, (from an Ever specification with an 2 element NAccessPoint array), we add more possible behaviours to describe the underlying link:

1. At any time, when there is an interaction from one ABP module to the other, the unreliable link may remove the interaction from the queue before it is seen by the peer ABP module.

See predicates *user_next* and *unreliable_link_next* in Appendix 4 for code that defines these actions of the environment.

From a version of Ever code produced without explicitly tying the NAccessPoint interaction points of the two ABP modules (Figure 11), more properties of an unreliable link can be modelled, namely delay, data duplication and data loss. An extra interaction queue is used between the pair of ABP modules in each direction as illustrated in the following diagram.

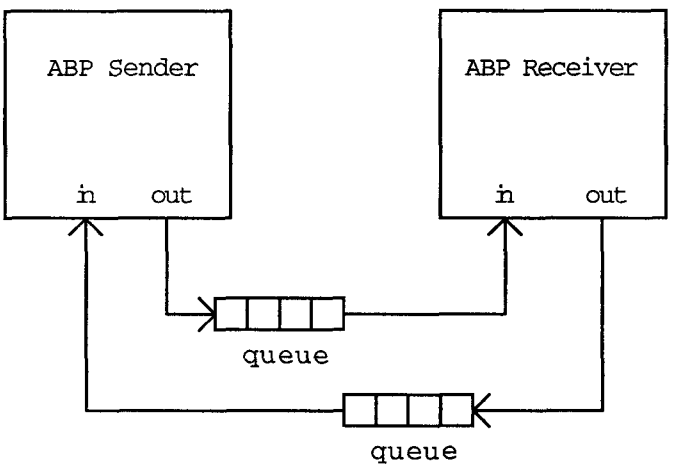


Figure 11. Alternating Bit Protocol module configuration for modelling unreliable underlying link that can lose, duplicate and repeat packets but not reorder packets

To do this we add the following possible behaviours to the actions of the underlying link:

1. Whenever one of the intermediate queues is full and the queue of the ABP module downstream is empty, the unreliable link may transfer the data from the intermediate queue to the next one downstream (i.e. complete a reliable transfer of a data packet), or

duplicate the data in the intermediate queue and place the duplicate in the next queue downstream (i.e. duplicate the data packet in transit).

2. Whenever one of the intermediate queues is full, the unreliable link may dequeue the data in the queue (i.e. model data loss).

Next we must specify a set of assumptions to restrict the set of possible behaviours to be considered by the model checker. We wish it to consider only fair executions, i.e. ones where the sending user regularly sends data, the receiving user regularly receives data, the unreliable link regularly transfers data (i.e. does not stay disconnected forever), and the timer clock ticks regularly. All these assumptions are specified as fairness constraints.

To prove the specification always delivers messages to the receiver in the same order as they are submitted, we prove properties about the ordering of various internal events of the specification. We will show that this applies to all sizes of data packets.

First, let us define the events used in the formulae that we are going to verify. Note an abbreviated form analogous to a WITH statement in the Pascal programming language is used in the formulae given with these events.

USend(x): true when a SENDrequest has been submitted by the sending user and is awaiting processing by the

ABP send module and the datum value of this request is x. The formula for this is:

$$UAccessPoint\_out[0].(queued = 1 \wedge content[0].kind = 0 \wedge SENDrequest.udata = x)$$

USendConfirm(): true when a SENDconfirm has been submitted by the ABP sender module and is awaiting processing by the sending user. The formula is:

$$UAccessPoint\_out[0].(queued = 1 \wedge content[0].kind = 1)$$

ASend(x,s): true when data packet containing datum value x and sequence number s has been submitted for transmission by the ABP sender module and is awaiting transfer over the underlying link. The formula (for the case with a reliable underlying link) is

$$NAccessPoint[1].(queued = 1 \wedge content[0].DATAinteraction.ndata.(id = 0 \wedge conn = 0 \wedge data = x \wedge seq = s))$$

ASendNothing(): true when ABP sender module has no request to transmit a packet over underlying link pending.

$$NAccessPoint[1].queued = 0$$

ARcv(x,s): true when a data packet containing datum value x and sequence number s is delivered to the ABP receiver module by the underlying link and has not yet

been processed by the ABP receiver module. The formula is the same as ASend(x,s), namely:

$$NAccessPoint[1].(queued = 1 \wedge content[0].DATAinteraction.ndata.(id = 0 \wedge conn = 0 \; data = x \wedge seq = s))$$

Note this formula is not accurate in the model of a unreliable link without two extra queues (first diagram above) because data loss can be modelled by removing an interaction in this queue after it has arrived in the receive queue of the ABP receive module. However this inaccuracy does not cause inaccuracies in the verification done below because the verified formulae specify that the interaction must stay in the queue until the ABP receiver module dequeues it. This formula would be a completely accurate representation of the ARcv event in the model with two extra queues in the data link (second diagram above).

ARcvNothing(): true when there is no packet pending receipt by the ABP receiver module in its queue from the underlying link:

$$NAccessPoint[1].queued = 0$$

AAcksend(s): true when the ABP receiver module has submitted a request to transmit an acknowledgement

packet containing sequence number s. The formula for this is:

$$NAccessPoint[0].(queued = 1 \wedge content[0].DATAinteraction.ndata.(id = 1 \wedge conn = 0 \wedge seq = s))$$

AAckrcv(s): true when an acknowledgement packet containing sequence number s has been delivered to the ABP sender module by the underlying link. Note this formula is inaccurate for the simpler unreliable link model in the same way as ARcv(x,s). The formula is:

$$NAccessPoint[0].(queued = 1 \wedge content[0].DATAinteraction.ndata.(id = 1 \wedge conn = 0 \wedge seq = s))$$

URcv(x): true when a RECEIVEresponse interaction containing datum value x has been submitted by the ABP receiver module and has not been processed by the receiving user module. The formula is:

$$UAccessPoint\_out[1].(queued = 1 \wedge content[0].(kind = 0 \wedge RECEIVEresponse.udata = x))$$

URcvreq(): true when a RECEIVErequest has been submitted by the receiving user and is awaiting processing by the ABP receive module. The formula is:

$$UAccessPoint\_in[1].(queued = 1 \wedge content[0].kind = 1)$$

Our goal is to show that this specification of the ABP protocol has the following properties when certain assumptions of its environment are made. We want to show that when the

user sends two requests, the receiving user will receive the same two requests in the same order. The assumption of the environment that must be made is that the underlying link is connected sufficiently often for long enough to transfer a data packet reliably.

The ideal formula to prove is of the form:

$$\forall x, y. AG(USend(x) \wedge nextUSend(y) \supset AF(URcv(x) \wedge nextURcv(y)))$$

A weaker form of this formula, namely

$$\forall x, y. AG(send(x) \wedge nextsend(y) \supset EF(rcv(x) \wedge nextrcv(y)))$$

was model checked. Unfortunately, one can only show directly this weaker liveness property. It states that if there exists a computation path such that the user sends two requests then there exists a computation such that those same two requests will be later received. The problem with this statement is that the two messages received can be a consequence of different send requests from the ones intended because it is assumed that the sending user keeps sending random data send requests.

The problem with checking the stronger liveness property is that the antecedent (containing universal quantification operators) is never true in the model of the system. It is false because the ABP Ever specification has too much non-determinism. It is never possible to write a formula to be true on all execution paths in this model that states that a given pair of send data requests are sent by the sending user.

This is not possible because the specification of the sending user's data is non-deterministic. Since this antecedent is false, the implication would be true for the wrong reason.

A considered solution was to change the actions of the sending user to always know the next two data values to be sent. But this idea is unnatural and unrealistic.

This type of problem may not have occurred if theorem proving was being used. In theorem proving, one states a set of assumptions and attempts to derive the goal from the assumptions using axioms and inference rules. In model checking, one cannot state an assumption about the behaviour of the model and ignore all its other behaviours because all formulae are evaluated using the model's nextstate relation. In the problem at hand, the assumptions correspond to the proposition "when the sending user sends a particular pair of data requests". The goal corresponds to the proposition "an identical pair of data requests will be later delivered to the receiving user".

Therefore, one has to verify the protocol a piece at a time and use logical reasoning to tie all the pieces' results together to make an overall conclusion. This approach is applied to the alternating bit protocol below.

For each send data request *usend(x)* from the sending user, the sender (i.e. ABP sender module) will repeatedly send a data

packet for the current sequence number *Send_seq* until an acknowledgement for it is received at which time it is ready to process the next send data request. The sender will be in the ESTAB state only when it is not awaiting receipt of an acknowledgement. The sender module will only accept a data send request from the user while in the ESTAB state.

When the ABP receiver module is ready to receive a new data packet (i.e. when *Recv_buffer_empty=TRUE*),

1. an acknowledgement is sent for each data packet that is received.
2. when a data packet containing sequence number equal to *Recv_seq* is received, it is passed onto the receiving user and *Recv_seq* is incremented.

When the ABP receiver module is not ready to receive a new data packet (i.e. when *Recv_buffer_empty=FALSE*), received data packets are dequeued and no acknowledgements are sent[1].

The sending user repeatedly alternately sends a random data request to the ABP sending module, and waits for a *SENDconfirm* response from the ABP sending module. Some of the specified fairness constraints ensure that only cases of data being sent infinitely often are considered.

---

[1]Only the latest version of the Estelle.Y alternating bit protocol specification does have this properly. The original version derived from the ISO Estelle specification [ISO89] loaded each received data packet into the current buffer even when the previous request had not been received by the user.

The receiving user repeatedly alternately requests to receive a data packet from the ABP receiving module, and receives a data packet. Some of the fairness constraints ensure that only cases of the receiving user's behaviour where data is infinitely often requested are considered.

With all the formulae given below, we show that they are consequences of the initial state of the whole system. Namely, for each formula f below we want to show that $initial\_state \supset f$ is a tautology. The following abbreviations are used:

Abbreviation	**Full Ever variable name**
state	abp[0].state
Send_buffer_empty	abp[0].vars.Send_buffer_empty
Send_buffer_seq	abp[0].vars.Send_buffer_seq
Send_buffer_datum	abp[0].vars.Send_buffer_datum
Send_seq	abp[0].vars.Send_seq

Table 13. List of Abbreviations used in formulae describing properties of alternating bit protocol

Noted with each formula is the name which is used in the Ever command file listed in Appendix 4.

First, to show that the sending module is only in the ESTAB state when it is not busy sending a data packet, verify

$$AG((state = ESTAB) \supset ASendNothing())$$

Formula ag_state_eq_ESTAB_implies_ASendNothing

We also claim that the ABP send module holds a data packet in an internal buffer only while in the ACK_WAIT state,

$$AG((state = ACK\_WAIT) \equiv (Send\_buffer\_empty = FALSE)).$$

Formula ag_send_buffer_empty_equiv_estab_state

While the send buffer is full, (i.e. while in ACK_WAIT state), the sequence number of the packet being retransmitted is equal to Send_seq,

$$AG(\neg Send\_buffer\_empty \supset (Send\_buffer\_seq = Send\_seq)).$$

Formula ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq

The ABP sender module is always in the ESTAB state or the ACK_WAIT state,

$$AG((state = ESTAB) \vee (state = ACK\_WAIT))$$

Formula ag_state_estab_or_ackwait

We claim that the ABP sender module never deadlocks under the specified fairness constraints,

$$AG(AF(state = ESTAB) \wedge AF(state = ACK\_WAIT))$$

Formula ag_af_state_ESTAB_and_af_state_ACKWAIT

If a send data request is submitted while in the ACK_WAIT state (i.e. while processing a previous request), it will remain in the request queue until at least the time at which the state returns to ESTAB.

$$\forall x.\, AG(USend(x) \wedge (state = ACK\_WAIT) \supset A[USend(x)\ U\ (USend(x) \wedge (state = ESTAB))])$$

Formula forall_x_one_send_request_at_a_time

In other words, the request is not processed while in the ACK_WAIT state; it is held in the queue until the state returns to ESTAB. When a request is in the queue while in the ESTAB state, a corresponding packet will be repeatedly transmitted until an acknowledgement is received.

$$\forall x.\, AG((state = ESTAB) \supset A[(ASendNothing() \wedge (state = ESTAB))\ U\ A[((ASendNothing() \vee ASend(x, Send\_seq)) \wedge (state = ACK\_WAIT))\ U\ (state = ESTAB)]])$$

Formula forall_x_repeat_transmit_packet

When an acknowledgement is received for current sequence number while in ACK_WAIT state, the state will be eventually returned to ESTAB and the current sequence number will be incremented.

$$\forall s, x.\, AG(((state = ESTAB) \wedge (s = Send\_seq) \wedge AAckRcv(s) \wedge (x = Send\_buffer\_datum)) \supset A[(ASend(x,s) \vee ASendNothing())\ U\ ((state = ESTAB) \wedge (Send\_seq = (s+1)))])$$

Formula forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab

In each cycle of the sending ABP module going from state ESTAB, ACK_WAIT and back to ESTAB, only the following events occur over the underlying link.

$$\forall x, s.\, AG((USend(x) \wedge (state = ESTAB) \wedge (s = Send\_seq)) \supset A[((ASend\_Nothing() \vee AAckRcv(s-1)) \wedge (state = ESTAB))\ U\ A[(ASend(x,s) \vee ASendNothing() \vee ARcvNothing() \vee ARcv(x,s) \vee AAckSend(x) \vee AAckRcv(s))\ U\ ((state = ESTAB) \wedge (Send\_seq = (s+1)))]]$$

Formula forall_x_s_send_cycle[2]

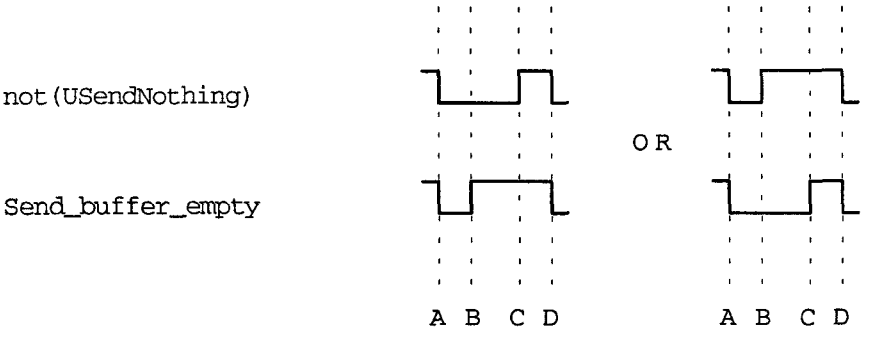
We have already shown that exactly one send request from the user corresponds to each ESTAB->ACK_WAIT->ESTAB state cycle. However, we have not shown that the Send_seq variable is incremented exactly once for each main state cycle. This is included in the discussion below.

Now, we show that each send request from the user corresponds to exactly one receive request at the other end. This is done in four steps from the sending user through the underlying link onto the receiving user at the other end. First, we show that the variable *Send_buffer_datum* is filled and emptied once for each send data request. Second, we show that the send sequence number variable *Send_seq* is incremented exactly once for each time the send buffer is filled and emptied. Third, we show that the receive sequence number

[2]Previous send data packet may be retransmitted after an acknowledgement for it has been received if it was sent to the underlying link before the acknowledgement was received and was still pending servicing by the underlying link when the acknowledgement was processed.

Recv_seq is incremented exactly once for each time the receive buffer is filled and emptied in the ABP receiver module. Also we show that the send sequence number is always equal to the receive sequence number at the instant that the receive buffer is filled. This shows that the send and receive sequence number match provided they match initially. Fourth, we show that there is exactly one receive request from the user for each time the receive buffer is filled and emptied.

In the ABP sender module, the send data buffer is always filled at exactly the same instant that the user's request is dequeued from the input interaction queue at PCO NAccessPoint. (This corresponds to the action of transition 1 in the Estelle.Y specification.) The times at which the next user request is enqueued in the NAccessPoint PCO and at which the send data buffer is emptied can be anytime before the next user request is dequeued. The possible behaviours are illustrated in Figure 13.



Instants:

A: user request dequeued and send data buffer filled.

B: either send data buffer emptied or another user request enqueued.

C: event opposite to that which occured at B.

D: start of new cycle, next user request dequeued and send buffer filled.

Intervals:

A-B: user request in buffer waiting for an ack. and no new user request yet.

B-C: either previous user request processed (ack received) and waiting for next user request or
next user request pending and still waiting for ack for previous user request.

C-D: next user request pending and previous request processed.

Figure 12. Interval between successive user send requests in Alternating Bit Protocol

Now the formula *usend_matches_send_buffer_fill* says that whenever there is a user send request pending, the following sequence of events must occur:

1 There may exist an interval of time from now to a point in the future such that the user request remains pending and the send buffer is full. The length of this interval can be zero.

2 There must be an interval of length of at least one time unit through which the user request still

remains pending and the send buffer is empty. This corresponds to the interval before A in Figure 13.

3 There must exist an interval from A to B of length of at least one time unit in which there is no pending user request and the send data buffer is full.

4 There may exist an interval between B and C (length zero or more) in which there is no new user request pending and the send buffer is empty (present request processed) or a new user request is pending and send buffer is still full (present request still being processed).

5 Finally there must be an interval from C to D of at least one time unit in length in which a new user request is pending and the send buffer is empty. In this interval the previous request has been completely processed and is waiting to process the next request (already pending). Immediately following this interval at D, there must be an instant at which the new user request has been dequeued and the send buffer is full.

The formula specifying that this sequence must occur is:[3]

$$
\begin{aligned}
&AG(\neg USendNothing() \supset \\
&\quad A[\neg USendNothing() \wedge \neg Send\_buffer\_empty\ U \\
&\quad\quad (\neg USendNothing() \wedge Send\_buffer\_empty \wedge \\
&\quad\quad\quad A[\neg USendNothing() \wedge Send\_buffer\_empty\ U \\
&\quad\quad\quad\quad (USendNothing() \wedge \neg Send\_buffer\_empty \wedge \\
&\quad\quad\quad\quad\quad A[USendNothing() \wedge \neg Send\_buffer\_empty\ U \\
&\quad\quad\quad\quad\quad\quad A[(USendNothing() \wedge Send\_buffer\_empty \vee \\
&\quad\quad\quad\quad\quad\quad\quad \neg USendNothing() \wedge \neg Send\_buffer\_empty)\ U \\
&\quad\quad\quad\quad\quad\quad\quad (\neg USendNothing() \wedge Send\_buffer\_empty \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\quad A[\neg USendNothing() \wedge Send\_buffer\_empty\ U \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad (USendNothing() \wedge \neg Send\_buffer\_empty)])]])])])
\end{aligned}
$$

Formula usend_matches_send_buffer_fill

The send sequence number variable, *Send_seq*, always contains the sequence number of the current data packet. While in the *ESTAB* state, this number is the sequence number that will be assigned to the next send data request received from the sending user. While in the *ACK_WAIT* state, this number is the sequence number of the data packet currently being retransmitted. Formula *send_buffer_fill_sequence* below specifies that this sequence number must remain constant throughout the send cycle except at the instant when a corresponding acknowledgement is received and the send buffer in the ABP sender module is emptied. In Figure 13, point B is the time at which this sequence number is incremented (i.e. when the *Send_buffer_empty* variable becomes true).

[3]Note USendNothing is equivalent to $\exists x.USend(x)$.

$$
\begin{aligned}
&AG(Send\_buffer\_empty \supset \\
&\quad \exists s.A[Send\_buffer\_empty\ U \\
&\qquad (\neg Send\_buffer\_empty \wedge (Send\_seq = s) \wedge \\
&\qquad\ A[\neg Send\_buffer\_empty \wedge (Send\_seq = s)\ U \\
&\qquad\quad (Send\_buffer\_empty \wedge (Send\_seq = (s+1)) \wedge \\
&\qquad\quad\ A[(Send\_buffer\_empty \wedge (Send\_seq = (s+1)))\ U \\
&\qquad\quad\quad (\neg Send\_buffer\_empty \wedge (Send\_seq = (s+1)))])])])
\end{aligned}
$$

Formula send_buffer_fill_matches_seq_number

The receive sequence number variable in the ABP receiver module, *Recv_seq*, always contains the sequence number of the next expected data packet to be received. Whenever a data packet with this sequence number is received and the receive buffer is empty, as indicated by the variable *Recv_buffer_empty*, the datum from the packet is loaded into *Recv_buffer_datum* and the receive sequence number is incremented. At all other times throughout the receive cycle, the receive sequence variable remains constant. Formula *receive_buf_filled_matches_seq_no* specifies this property.

$$
\begin{aligned}
&AG(Recv\_buffer\_empty \supset \\
&\quad \exists s.A[Recv\_buffer\_empty\ U \\
&\qquad (\neg Recv\_buffer\_empty \wedge (Recv\_seq = s) \wedge \\
&\qquad\ A[(\neg Recv\_buffer\_empty \wedge (Recv\_seq = s))\ U \\
&\qquad\quad (Recv\_buffer\_empty \wedge \\
&\qquad\quad\ A[(Recv\_buffer\_empty \wedge (Recv\_seq = s))\ U \\
&\qquad\quad\quad (\neg Recv\_buffer\_empty \wedge (Recv\_seq = (s+1))])])])
\end{aligned}
$$

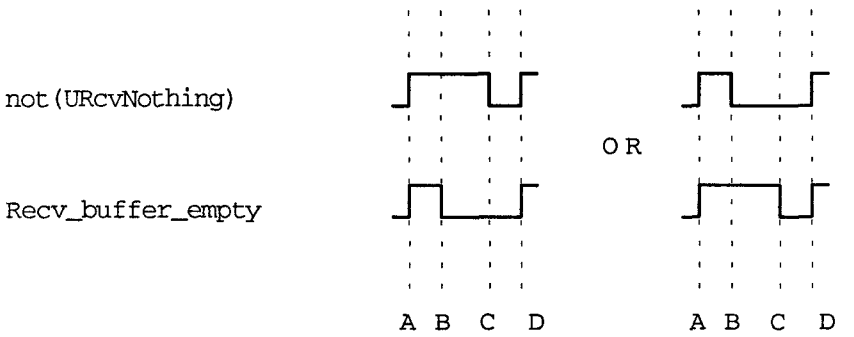
Formula receive_buf_filled_matches_seq_no

In the ABP receiver module, the receive buffer, *Recv_buffer_datum*, is always emptied at exactly the same instant the received datum is enqueued into the interaction

queue to the receive user. Each receive cycle starts at this instant and is followed by the following intervals:

1. a period of at least one time unit in which the received datum is still pending receipt by the user and the receive buffer in the ABP receiver module is empty.
2. a (optional) period of any length in which either the received datum is still awaiting dequeuing by the user and the receive buffer has been filled, or the received datum has been dequeued and the receive buffer is still empty awaiting the next data packet.
3. a period of at least one time unit in which the user interaction queue is empty (ready for delivery of the next data packet) and the receive buffer is full (i.e. the next data packet has already been received).

These intervals are illustrated in Figure 13.



Instants:

A: received data is delivered to user and receive buffer is emptied.

B: either another data packet is received or user has dequeued previous data interaction

C: event opposite to that which occurred at B.

D: start of new cycle, next received data packet is delivered to user and receive buffer is emptied

Intervals:

A-B: received data packet in user interaction queue and waiting for next data packet from underlying link.

B-C: either received packet still pending receipt by user and next data packet received or
user dequeued received packet and still waiting for next data packet from underlying link.

C-D: user ready for delivery of next data packet and next data packet already received from underlying link.

Figure 13. Interval between successive user receive requests in Alternating Bit Protocol

$$
\begin{aligned}
&AG(URcvNothing \supset \\
&\quad A[(URcvNothing \wedge Recv\_buffer\_empty)\ U \\
&\quad\quad (URcvNothing \wedge \neg Recv\_buffer\_empty \wedge \\
&\quad\quad\quad A[(URcvNothing \wedge \neg Recv\_buffer\_empty)\ U \\
&\quad\quad\quad\quad (\neg URcvNothing \wedge Recv\_buffer\_empty \wedge \\
&\quad\quad\quad\quad\quad A[(\neg URcvNothing \wedge Recv\_buffer\_empty)\ U \\
&\quad\quad\quad\quad\quad\quad A[((URcvNothing \wedge Recv\_buffer\_empty) \vee \\
&\quad\quad\quad\quad\quad\quad\quad (\neg URcvNothing \wedge \neg Recv\_buffer\_empty))\ U \\
&\quad\quad\quad\quad\quad\quad\quad (URcvNothing \wedge \neg Recv\_buffer\_empty \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\quad A[(URcvNothing \wedge \neg Recv\_buffer\_empty)\ U \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad (\neg URcvNothing \wedge Recv\_buffer\_empty)])]])])])
\end{aligned}
$$

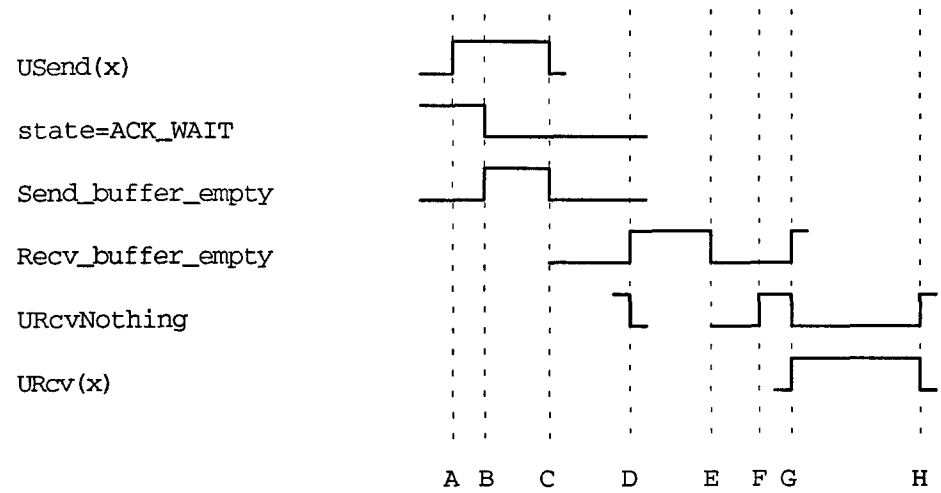
Formula urcv_matches_recv_buff_fill

We have shown that each of the four phases above always corresponds to a single send data request. Now to show that the control structure of the protocol is correct for any packet data size, we must show that the data in a send data request is always preserved from the time the request is submitted to the time it is received by the receiving user. This is specified in formula *forall_x_data_preserved*. It says that whenever a send request is submitted by the sending user, the following series of intervals illustrated in Figure 14 must occur:

1. An optional interval in which ABP sender state is ACK_WAIT and user's send request is still pending servicing. This interval only occurs when the user's send request is submitted while the ABP sender module is processing a previous request. This interval corresponds to A-B in Figure 14.

2. An interval of at least one time unit in which the ABP sender state is ESTAB and the user's send request is pending servicing. This corresponds to B-C in Figure 14.
3. An interval that must start with a time unit in which the ABP sender state is ACK_WAIT and the user's send request has been transferred to the send buffer *Send_buffer_datum*. During this interval, the previous and current send request packets are being sent and received. It must end with a phase in which the ABP receiver module receive buffer *Recv_buffer_datum* is empty. This interval ends when the ABP receiver module fills its receive buffer with the current send data packet. This corresponds to C-E in Figure 14.
4. This interval corresponds to the period between when the ABP receiver's receive buffer is filled and the time at which the data packet is delivered to the receive user. The ABP receiver's receive buffer is full throughout this interval. It ends with a phase in which the interaction queue to the user is empty. It may begin with a phase in which this interaction queue is still filled with the previous data packet awaiting the user to retrieve it. This corresponds to E-G in Figure 14.

5. In the last interval, the ABP receiver's receive buffer is empty and the received packet is in the interaction queue to the user awaiting the user to dequeue it. This is G-H in Figure 14.



Instants:

A: user submits a request to send datum x
B: ABP sender module finished sending previous request
C: ABP sender starts processing current request
D: ABP receiver's receive buffer becomes empty
(i.e. transfers previous packet to user's interaction queue)
E: current packet loaded into ABP receiver's receive buffer
F: user has dequeued previous packet from interaction queue
G: current packet is loaded into user's interaction queue
H: user has dequeued current packet

Intervals:

*A-B: user send request submitted while ABP sender module busy processing previous request.
B-C: user request submitted, ABP sender module idle, waiting for ABP sender to start processing request.
*C-D: data packet being retransmitted while ABP receiver module receive buffer is full containing previous packet.
D-E: data packet being retransmitted while ABP receiver module receive buffer is now empty.
*E-F: data packet received by ABP receive module into its receive buffer and previous packet awaiting user to dequeue it.
F-G: data packet received by ABP receive module and interaction queue to user empty.
G-H: data packet loaded into interaction queue to user, waiting for user to dequeue it.

Intervals marked with asterisks may have zero length.

Figure 14. Interval from time data request submitted by send user to time at which it is received by receive user.

$$
\begin{array}{l}
\forall x. \\
AG(USend(x) \supset \\
\quad A[((state = ACK\_WAIT) \wedge USend(x))\ U \\
\quad\quad ((state = ESTAB) \wedge USend(x) \wedge \\
\quad\quad\ A[((state = ESTAB) \wedge USend(x))\ U \\
\quad\quad\quad ((state = ACK\_WAIT) \wedge USendNothing \wedge (Send\_buffer\_datum = x) \wedge \\
\quad\quad\quad\ A[((state = ACK\_WAIT) \wedge \neg Recv\_buffer\_empty \wedge \\
\quad\quad\quad\quad (ASendNothing \vee ASend(x, Send\_seq) \vee \exists y.\, ASend(y, Send\_seq - 1)))\ U \\
\quad\quad\quad\quad (Recv\_buffer\_empty \wedge (ARcvNothing \vee ARcv(x, Send\_seq) \vee \\
\quad\quad\quad\quad\quad\quad\quad\quad \exists y.\, ARcv(y, Send\_seq - 1)) \wedge \\
\quad\quad\quad\quad\ A[(Recv\_buffer\_empty \wedge \\
\quad\quad\quad\quad\quad (ARcvNothing \vee ARcv(x, Send\_seq) \vee \\
\quad\quad\quad\quad\quad\ \exists y.\, ARcv(y, Send\_seq - 1)))\ U \\
\quad\quad\quad\quad\quad (\neg Recv\_buffer\_empty \wedge (Recv\_buffer\_datum = x) \wedge \\
\quad\quad\quad\quad\quad\ A[(\neg Recv\_buffer\_empty \wedge (Recv\_buffer\_datum = x) \wedge \\
\quad\quad\quad\quad\quad\quad \exists y.\, URcv(y))\ U \\
\quad\quad\quad\quad\quad\quad (\neg Recv\_buffer\_empty \wedge (Recv\_buffer\_datum = x) \wedge \\
\quad\quad\quad\quad\quad\quad\ URcvNothing \wedge \\
\quad\quad\quad\quad\quad\quad\ A[(\neg Recv\_buffer\_empty \wedge \\
\quad\quad\quad\quad\quad\quad\quad (Recv\_buffer\_datum = x) \wedge URcvNothing)\ U \\
\quad\quad\quad\quad\quad\quad\quad (Recv\_buffer\_empty \wedge URcv(x) \wedge \\
\quad\quad\quad\quad\quad\quad\quad\ A[URcv(x)\ U\ URcvNothing])])])])])])])
\end{array}
$$

Formula forall_x_data_preserved

5.2. Results of experiments on alternating bit protocol

This section discusses the results obtained from running the formulae in the previous section through the model checker with the model containing an unreliable underlying link which only delivers and loses packets. All the formulae evaluated as tautologies confirming the truth of the corresponding properties in the alternating bit protocol specification.

Section 5.2.1. reports figures on the time and memory resources used to compute the formulae in the previous section. The main result is that the system did not suffer from the

state explosion problem. Some explanation on why the evaluation of some formulae took longer than others is given.

Section 5.2.2. proposes a method for generating counterexamples for some classes of CTL formulae. In section 5.2.3. this method is applied to the first formula *ag_state_eq_ESTAB_implies_ASendNothing* which was originally specified incorrectly. A corrected formula is proposed and shown to verify as correct.

For detailed information on the verification, refer to the appendices. The alternating bit protocol specification written in Estelle.Y and ASN.1 is given in Appendices 1 and 2, respectively. Appendix 3 shows the Ever code produced by the *estelle2ever* translator. The Ever command file for carrying out the verification of these formulae is given in Appendix 4. Actual output from the model checker is shown in Appendix 5. Appendix 4 contains Ever commands for:

1. defining the actions of the user modules (sender and receiver)
2. defining the actions of the modelled underlying unreliable link.
3. defining the set of fairness constraints
4. defining the global nextstate relation
5. setting the default nextstate relation and fairness constraints to be used by all formulae being verified

6. specifying and verifying properties discussed in the previous section.

5.2.1. Resource Usage

Table 14 gives the figures of time and memory used to verify each of the formulae discussed in section 5.1. For more detailed statistics see the appendices.

Formula Name	CPU time (mins)	Memory at completion (kb)
ag_state_eq_ESTAB_implies_ASendNothing	57:18	4157
ag_send_buffer_empty_equiv_estab_state	0:06	4157
ag_not_Send_buffer_empty_implies_Send_buffer_seq_ eq_Send_seq	0:05	4157
ag_state_estab_or_ackwait	0:01	4157
ag_af_state_ESTAB_and_af_state_ACKWAIT	61:01	4157
forall_x_one_send_request_at_a_time	94:39	4189
forall_x_repeat_transmit_packet[4]	184:38	4189
forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab	251:57	4253
forall_x_s_send_cycle	404:20	7197
usend_matches_send_buffer_fill[5]	353:34	4253
send_buffer_fill_matches_seq_number	506:02	7133
receive_buf_filled_matches_seq_no	576:16	12765
urcv_matches_recv_buff_fill	466:20	12829
data_preserved	2412:22	24477

Table 14. CPU Time and Memory used during Verification

[4]The test was restarted from here, so the accumulated CPU time figures for subsequent formulae are higher than if test was run all at once. They could be exagerated by as much as the time for formula *ag_state_eq_ESTAB_implies_ASendNothing* (the smaller of *ag_state_eq_ESTAB_implies_ASendNothing* and *forall_x_repeat_transmit_packet*) because tests *ag_state_eq_ESTAB_implies_ASendNothing* and *forall_x_repeat_transmit_packet* include time for computing *EG True* for fairness constraints.

[5]A new test was started here, so this includes computation of *EGTrue*.

The main result is that all these formulae were evaluated in finite time and memory. All the formulae evaluated as tautologies confirming that all the formulae are true in the alternating bit protocol specification thus allowing the argument given in section 4.1 to be used to prove the protocol correct for all packet sizes.

Formulae *ag_send_buffer_empty_equiv_estab_state*, *ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq* and *ag_state_estab_or_ackwait* took very little time because they are safety properties. For these the model checker only needs to check whether these properties are maintained from one time unit to the next. These formulae are all of the form AGf. The dual of this formulae, namely $\neg EF\neg f$, is evaluated. With fairness constraints, this is evaluated as $\neg EF(\neg f \wedge Fair)$ without using the fairness constraints where *Fair* was previously evaluated as *EGTrue* with the fairness constraints. This expression is evaluated as $\neg E[True\ U\ (\neg f \wedge Fair)]$ which is computed as the least fixed point of $Z = (\neg f \wedge Fair) \vee EXZ$. If f is a valid safety property, this fixpoint computation will only take one iteration because by definition a safety property is maintained from one time unit to the next.

Formula *ag_af_state_ESTAB_and_af_state_ACKWAIT* took far longer than the previous formulae because it required a complete analysis of all possible execution paths in the system

to determine whether either state can always be reached in the future.

5.2.2. Method to generate counterexample traces

The system discussed so far implements a facility for evaluating CTL formulae in a given model. A formula evaluates to a Boolean characteristic function of the set of states (combinations of all Ever variable values in current phase) that satisfy the formula in the current time unit. Usually during verification, one wishes to confirm that a given CTL formula is a tautology when evaluated in the model. And if not, one would prefer to see a trace of an execution path that violates the formula (to assist the designer find the flaw in the formula or the specification).

Some proposed procedures for producing counterexamples for formulae of a few syntactic forms are described below in terms of invoking Ever's *printtrace* facility.

Recall, when verifying properties of a finite state machine based system, one is interested in considering only execution paths that start in an initial state. Thus formulae to be evaluated are often of the form $initial\_state \supset f$. If the model checker produces a tautology, one is finished. Otherwise, one should evaluate $f \wedge initial\_state$ to see if f is satisfiable in the model (i.e. true in any of the initial states).

Let *nextstate* denote the nextstate relation of the finite state machine. Recall, the parameters for the printtrace command are: *initial,* a proposition defining the set of initial states; *nextstate,* the nextstate relation of the model; and *goal,* a proposition defining a set of states. Printtrace attempts to find a path from any state in the initial states to any state in the goal set using the given nextstate relation. Note *printtrace* will always find such a path for the types of formulae discussed below because we only use it when the original formula failed to evaluate as a tautology.

Suppose formula f is of the form AGg with g non-temporal. This form is typical of safety properties. If AGg is false, then its complement (defined by the duality properties of CTL) $EF\neg g$ must be true. This formula is interpreted as: there exists a path to a state in which $\neg g$ is true. One can produce such a path with *printtrace TRUE nextstate* $\neg g$.

Note one would want to show a formula of the form AGg is a tautology only in a system intended to start in any possible state. To find a counterexample of a safety property AGg in a system with initial states *init,* one is to find a counterexample of formula $init \supset AGg$. Since formula $init \supset AGg$ is false when $init \wedge \neg AGg = init \wedge EF\neg g$ is true (property of implications, CTL duality properties), such a counterexample is a trace to $\neg g$ from any state satisfying proposition *init*. To do this, we change the set of initial

states passed to the printtrace command, namely *printtrace init nextstate* $\neg g$.

The printtrace routine could be modified to handle the more general case with g temporal. The phase of printtrace that repeatedly calculates forward images could be extended to apply this counterexample algorithm again when a goal state is reached. In this nested application of this algorithm the start state would be the set of states just reached and the goal set would be defined by the temporal formula g. For example, consider formula $f \supset AG(g \supset AFh)$. First, this algorithm is applied to find a trace from a state satisfying f to a state satisfying $\neg(g \supset AFh)$. Then the algorithm is called again to find a trace from the state reached so far to one violating $g \supset AFh$, namely $g \wedge \neg AFh$. The algorithm described below for AU operators would be used since $AFh \equiv A[True\ U\ h]$.

Suppose formula f is of the form $A[g\ U\ h]$. To produce a counterexample of this formula, one has to produce a trace satisfying the properties expressed by the complement of $A[g\ U\ h]$. The CTL duality properties state $A[g\ U\ h] = \neg E[\neg h\ U\ (\neg g \wedge \neg h)] \wedge \neg EG\neg h$. By DeMorgan's Boolean logic laws, the complement of this expression is $E[\neg h\ U\ (\neg g \wedge \neg h)] \vee EG\neg h$. This says that a counterexample would be:

- any trace that leads to a state with $\neg g \wedge \neg h$ true and has h false in all preceding states.

- any infinite trace with h false throughout

To support finding a trace of the first type above, a copy of the printtrace routine would need to be modified. The difference between the original printtrace routine and the new one would be that at each step (each time a forward image is computed), the set of states would be restricted to those in the computed set satisfying formula $\neg h$. This new routine would be invoked with $\neg g \wedge \neg h$ specified as the goal.

To support finding a trace of the second type above, a copy of the printtrace routine would need to be modified in a similar way. In each step the set of states would be restricted to those in the original method's computed set satisfying $\neg h$. This routine would also have to detect loops in the state graph. Once one such loop is found, it can be used as a counterexample trace.

It should be possible to combine these two algorithms into a single routine which repeatedly computes forward images for both types of traces.

This new routine should be invoked with start state *TRUE* for formulae f of the form $A[g\ U\ h]$ and with start state init for formulae f of the form $init \supset A[g\ U\ h]$.

To produce a counterexample trace for a formula f of form AXg or $init \supset AXg$, one can evaluate the negation of these formulae with the model checker and derive the first state of the two state trace from the BDD output.

To produce a counterexample trace for a formula of the form *AFg*, the definition *AFg = A[True U g]* can be applied.

So far methods for producing counterexamples of temporal formulae with universal quantification have been discussed. The duality properties define the negation of universally quantified formulae to be existential formulae. The printtrace algorithm can only be applied to these because only one path satisfying a given goal need be found.

To find a counterexample of an existentially quantified temporal formula would require showing that all possible paths satisfy a given property.

Fortunately when verifying systems, one is usually only interested in verifying properties that are true in all possible execution paths

For all of the above algorithms, the printtrace should be modified to consider the fairness constraints when computing forward images to ensure unfair paths are ignored.

The above algorithm for formulae of the form *init* $\supset$ *AGg* is manually applied to one of the properties of the alternating bit protocol in the next section.

5.2.3. Application of Counterexample Method

The original version of several of the formulae for properties of the alternating bit protocol given in section 5.1 evaluated to false in all the initial states. One of the

counterexample generating methods described in the previous section was applied to formula ag_state_eq_ESTAB_implies_ASendNothing specified in section 5.1. The counterexample points out a misunderstanding in the original specification of the formula. The intended meaning of the original formula was "whenever the ABP sender module is in the ESTAB state, the module never initiates a send request to the underlying data link." The formula was specified as:

```
AG((state = ESTAB) ⊃ ASendNothing) where
        ASendNothing ≡ (NAccessPoint[1].queued = 0)
```

The trace output in Appendix 6 for this formula is explained. Since the Ever code for the alternating bit protocol specification specifies a condition that all initial states must satisfy, the method for generating counterexamples for formulae of form *init* ⊃ *AGg* is used. A printtrace command with initial state *global_init*, nextstate relation *global_nextstate* and goal ¬*((state = ESTAB)* ⊃ *ASendNothing)* is used. Propositions *global_init* and *global_nextstate* are defined in the Ever command file (see Appendix 4). The trace output is in reverse order (i.e. the end of the trace (a state satisfying the goal) is printed first). For each step in the trace, the values of all the variables in their current phase are printed. The state at the beginning of each step in the trace is shown in terms of the abbreviations used in section 5.1 in the following table:

time unit	clock tick	state	USend (0)	USend Confirm	ASend (0,0)	AAckSend (0)	timer running, timer counter
0	F	ESTAB	F	F	F	F	F
1	F	ESTAB	T	F	F	F	F
2	T	ACKWAIT	F	T	T	F	T,0
3	T	ACKWAIT	F	T	T	F	T,1
4	T	ACKWAIT	F	T	T	F	T,2
5	F	ACKWAIT	F	T	T	F	T,3
6	F	ACKWAIT	F	T	F	T	T,3
7	F	ACKWAIT	F	T	T	T	T,0
8	F	ESTAB	F	T	T	F	F

Table 15. Analysis of a counterexample trace

In this case, the value of ASendNothing is the complement of ASend(0,0). At time unit 8, the invariant claimed by the formula is violated.

Time unit	Description of state before action	Action taken
0	Initial state	sending user submits a send data request to ABP sender
1	The ABP module is in its ESTAB state waiting for a request and there is a send request pending servicing in the interaction between the sending user and the ABP sender module.	ABP sender sends data packet to underlying data link (trans1)
2	The ABP sender has moved to the ACKWAIT state, sent a confirm message back to the sending user, requested the underlying data link to send a data packet, and has started its retransmit timer.	ABP sender timer tick.
3 and 4	Time has passed and the retransmit timer counter has been incremented.	ABP sender timer tick.
5	The retransmit timer's counter has reached its maximum value, the data packet has been delivered to the ABP receiver module.	ABP receiver receives data packet and sends an acknowledgement to the underlying data link (trans7).
6	The data packet has been removed off the data link by the ABP receiver module, the ABP sender retransmit timer is still expired awaiting processing, and an acknowledgement is in transit.	ABP sender retransmits data packet because timer has expired (trans4).
7	The second retransmission of the data packet is still in transit, the acknowledgement has been delivered into the ABP sender's receive interaction queue from the underlying link.	ABP sender receives the acknowledgement and returns to the ESTAB state.
8	The ABP sender module has received the acknowledgement packet and returned to the ESTAB state with the second retransmission of the data packet still in transit (contradicting the claimed invariant).	

Table 16. Descriptions of steps in counterexample trace

This trace shows that the correct interpretation of the original formula would have been "whenever the ABP sender module is in the ESTAB state, it never has a send request pending processing by the underlying data link.".

To express the intended property correctly, the formula must be modified to:

```
AG((state = ESTAB) ⊃
    A[((∃y,s.ASend(y,s) ∨
        A[((state = ESTAB) ∧ ASendNothing) U (state = ACK_WAIT)])) U
          (state = ACK_WAIT)])
```

This formula says that the if the state is now ESTAB, the following must be true until state becomes ACK_WAIT. During this interval, any send request to the underlying data link may be pending servicing or if no send request is pending no others may be submitted for the rest of the interval. In other words, this formula states that no new send request may be initiated while in the ESTAB state. The Ever command file and output file are shown in Appendices 7 and 8, respectively. Applying the new version of this formula to the model checker produced a tautology.

This section has demonstrated that the tool can be used to analyze a protocol specification and force the specifier to grasp a solid understanding of the specification and its properties.

Chapter 6: Conclusions and Future Work

6.1. Conclusions

A system has been developed for carrying out incremental verification of communication protocols. A complete set of internal properties of an Estelle.Y [Lu91] specification of the alternating bit protocol (containing over 10^{15} possible states) was successfully verified using less than 25 megabytes of memory and less than 100 hours CPU time on a Sun Sparc 10/15 computer.

This set of verified internal properties were combined with some reasoning to show that the structure of the protocol is valid for all packet data sizes.

A proposed method for generating counterexamples (not yet integrated into Ever) was applied to the original version of one of the formulae. It pointed out a misunderstanding in the formula's original specification. The formula was revised and confirmed to be a tautology by the verification system. This example demonstrated that this tool can be used for troubleshooting formulae and the protocol specification itself.

After one temporal property involving all modules in the specification has been verified, it appears that this system does not consume significant additional memory as each additional property is verified. This suggests that the size of

the specification is the key factor in determining the amount of memory required because once all the nextstate relation's partitions have been evaluated into BDDs they can be reused for later computations.

The author realized that model checking is limited to only checking properties that are strictly "observational" of the model being verified. One cannot use temporal formulae of the form "assumed input behaviour implies expected output behaviour" as one does in theorem proving and expect the actions associated with the "assumed input behaviour" to be added to the model during the verification. One has to build a model that includes all the behaviours that one wishes to treat as antecedents in implications to be verified. Then the model checking can restrict the sets of behaviours it considers.

The overall liveness property of the alternating bit protocol, stating that any given pair of send data requests always results in a pair of data packets with same data values being delivered to the receiving user in the same order (i.e. $\forall\ x1,x2.AG(send(x1) \wedge nextsend(x2) \supset AF(recv(x1) \wedge nextrecv(x2)))$), cannot be directly verified with model checking because it is impossible to produce a model that includes all possible behaviours of the antecedent of this formula. Each antecedent behaviour of this formula needs to be expressed with universal quantification operators to make the formula express the desired overall liveness property. It is not possible for

any model with a non-deterministic data source to satisfy the antecedent for every possible pair of data values at the same time. If a non-deterministic data source is not used, the model is not sufficiently general to verify the protocol. Because of this problem, the protocol had to be verified by combining model checking of the protocol's internal properties with reasoning about how the internal properties are related.

This system is suitable for verifying properties of a protocol whose environment's behaviour can be described by a simple state machine and a number of optional conditions that must occur infinitely often in the state machine. The alternating bit protocol environment was described by three simple state machines (for the sending and receiving users and the underlying link) and a set of fairness constraints limiting the set of behaviours considered during verification.

The complexity of the tableau construction algorithm is too large to be practical even for trivial formulae (e.g. reliable link specification).

Small maximum timer count values can be used in Estelle.Y specifications given to this system because the verification considers all possible timing interleavings of Estelle.Y module actions and timer tick actions. The maximum timer values only have significance when two or more timers are started in a Estelle.Y transition and they are specified to expire at different times. These small values allow the size of the

specification to be minimized (in terms of numbers of bits for timers in BDDs).

6.2. Future Work

A number of features that should be implemented to optimize the memory and CPU time usage of the Ever model checker are discussed.

Instantiations of Ever predicates (i.e. predicate name and actual parameters) should be cached so that unnecessary recomputation of predicates with same parameters is eliminated. Predicates with temporal operators can take a significant time to evaluate.

When a proposition with operands (e.g. *(and p1 ... pn)*) has been evaluated, the BDDs of its operands should be freed if no other unevaluated propositions refer to these operands. This would eliminate unnecessary use of memory.

The algorithms discussed in chapter 5 for producing counterexample traces should be implemented.

The Estelle.Y language should be extended to allow the explicit specification of a size for each internal variable. For example, the LAPB protocol uses 3-bit sequence numbers. The 1-bit sequence number variables in the alternating bit protocol were declared as Boolean variables to avoid unnecessary use of BDD bit variables in the nextstate relation.

It would be convenient to the user if the Ever interface supported aborting of the evaluation of propositions without stopping the whole program. Since verification commands can take hours, it would be beneficial to be able to abort a command without losing the results of previous work when one realizes that a mistake was made in a command.

With a few changes to this system, the behaviour of the environment could be specified as Estelle.Y modules instead of having to be written directly in Ever. The Estelle.Y language would need to be extended with a new command for defining *free* variables to allow a module to introduce non-deterministic data into the model. Also, one should be allowed to define arrays of PCOs. For example, the underlying data link of the alternating bit protocol could be represented with a module that has two NAccessPoint PCOs, one tied to each of the ABP sender and ABP receiver modules.

Perhaps some scheme could be devised to allow fairness constraints to be automatically derived from information in the Estelle.Y modules defining the environment's behaviour.

A feature to automatically check for protocol independent properties such as deadlock, unreachable transitions and ambiguous transitions in Estelle.Y modules could be implemented. Presence of deadlock is represented with a formula of the form $\exists\, i.(init \supset EF(EG(state=i))).$

A transition is unreachable if its enabling condition is always false i.e.

$init \supset AG(\neg\ transition\_condition)$.

There exists unreachable transitions if

$\exists\ i.(init \supset AG(\neg\ transition_i\_condition))$.

A module has ambiguous transitions (i.e. is non-deterministic) if

$\exists\ i,j.(init \supset AF(transition_i\_condition \wedge\ transition_j\_condition))$.

As a notational convenience to the Ever user and the Estelle.Y translator, the *forall* and *exists* operators should be implemented. In the Ever command files for verifying the alternating bit protocol, the *exists* operator was manually specified as a disjunction and the *forall* operator was manually specified as a conjunction. A new notion of metavariables should be introduced into Ever so that expressions with quantified variables can be used without adding more variables to the BDD. Brace's BDD package includes a primitive for evaluating *exists* and *forall* operators with the quantifier variable being an existing BDD variable.

To reduce the number of bit variables produced in the BDD from the data structures for PCOs which logically contain arrays of variant records, Ever should be extended to support variant record types.

TESTGEN [VHLMD93] is a tool for generating test cases from an Estelle.Y protocol specification. The test case generation

is guided by a set of user-defined constraints. These constraints specify the values of data fields to be tested and the minimum and maximum number of times each *protocol element* shall occur in each generated test. A protocol element is defined as a major state, transition, constant, variable, input service primitive, output service primitive, or timer. The method of generating counterexamples in the Ever tool could be modified to produce execution traces associated with a given property expressed as a temporal logic formula. This would provide an interesting way to produce test cases associated with a particular function of a protocol.

Since the verification tool discussed in this thesis uses the Estelle.Y language and its internal protocol data structure (PDS), this tool could be integrated with TESTGEN in the overall integrated protocol engineering environment developed by the Protocol Engineering Group in the UBC Computer Science department.

BIBLIOGRAPHY

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking", 27th ACM/IEEE Design Automation Conference, pp. 46-51, 1990

[BCMDH90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond", Proceedings of the Fifth Annual Symposium on Logic in Computer Science, June 1990.

[BF89] S. Bose, A.Fisher, "Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic", IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, 1989.

[BRB90] K.S. Brace, R.L. Rudell, R.E. Bryant, "Efficient Implementation of a BDD Package", 27th ACM/IEEE Design Automation Conference, pp. 40-45, 1990.

[Bry86] R.E. Bryant, "Graph-based algorithms for Boolean function manipulation", IEEE Trans. Comput., C-35(8), 1986.

[Bur93] Jerry Burch, Stanford University, electronic mail correspondence.

[CBM89] O. Coudert, C. Berthet, J.C. Madre, "Verification of synchronous sequential machines based on symbolic execution", In J. Sifakis, editor, *Automatic Verification methods for Finite State Systems*, International Workshop, Grenoble, France, volume 407 of *Lecture Notes in Computer Science*, Springer-Verlag, June 1989.

[CE81] Clarke, Emerson, "Synthesis of synchronization skeletons for branching time temporal logic", Proc. Workshop on Logic of Programs, Yorktown Heights, NY: Springer-Verlag, pp. 52-71, 1981.

[CE81] E.M. Clarke, E.A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic", Logics of Programs Proceedings, Lecture Notes in Computer Science 131, 1981.

[CES86] Clarke, Emerson, Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", ACM Trans. Program. Lang. Syst. 892): 244-263, 1986.

[CG87] E.M. Clarke, O. Grumberg, "Research on Automatic Verification of Finite State Concurrent Systems", Ann. Rev. Comput. Sci. 2:269-290, 1987.

[CG89] E.M. Clarke, O Grumberg, "The Model Checking Problem for Concurrent Systems with Many Similar Processes", Temporal Logic in Specifications Proceedings, Lecture Notes in Computer Science 398, pp. 188-201.

[CLV93] S. Chanson, A. Loureiro, S. Vuong, "On Tools supporting the use of Formal Description Techniques in Protocol Development", Computer Networks & ISDN Systems, Vol. 25, pp. 723-739, 1993.

[CR85] Lori A. Clarke, Debra J. Richardson, "Applications of Symbolic Evaluation", The Journal of Systems and Software 5, 15-35 (1985).

[ES88] E.A. Emerson, J. Srinivasan, "Branching Time Temporal Logic" Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency School/Workshop, Noordwijkerhout, Norway, 30 May - 3 June 1988

[Fil91] Thomas Filkorn, "Functional Extension of Symbolic Model Checking", Proceedings of the Workshop on Computer-Aided Verification, July 1-4, 1991.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, "The temporal analysis of fairness", Proceedings 7th ACM Symposium on Principles of Programming Languages (Las Vegas, Jan. 1980), pp. 163-173.

[HDDY92] Alan J. Hu, David L. Dill, Andreas J. Drexler, C. Han Yang, "High-Level Specification and Verification with BDDs", Computer Science Department, Stanford University, 20 May 1992.

[HO83] Brent T. Hailpern, Susan S. Owicki, "Modular Verification of Computer Communication Protocols", IEEE Transactions on Communications, vol. Com-31, No. 1, January 1983, pp. 56-68.

[Hu92-93] Alan J. Hu, electronic mail correspondence.

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

[JPHS91] S.-W. Jeong, B. Plessier, G.D. Hachtel, F. Somenzi, "Variable Ordering for FSM Traversal", Proceedings of the International Workshop on Logic Synthesis, MCNC, Research Triangle Park, NC, May 1991.

[LP85] O. Lichtenstein, A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification", Conf. Rec. Twelfth Annu. ACM Symp. Principles Program. Lang., new Orleans, pp. 97-107., 1985.

[LS82] S.S. Lam, A.U. Shankar, "An Illustration of Protocol Projections", Protocol, Specification, Testing and Verification, 1982, pp. 343-360.

[Lu91] Ying Lu, "On TESTGEN, An Environment for Protocol Test Sequence Generation, And Its Application to the FDDI MAC Protocol", M.Sc. Thesis, Aug. 1991, University of British Columbia, Canada.

[MP4] Z. Manna, A. Pnueli, "Verification of Concurrent Programs: A Temporal Proof System", Foundations of Computer Science IV, J.W. Debakker, J. Van Leeuwen, Eds., Mathematical Center Tracts 159, Amsterdam (1983) 163-255.

[MP5] Z. Manna, A. Pnueli, "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs", Science of Computer Programming 4 (1984) 257-289.

[MWBA88] Sharad Malik, Albert R. Wang, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, "Logic Verification using BDDs in a Logic Synthesis Environment", Proc. Int. Conf. CAD (ICCAD-88), pp. 6-9, Nov. 1988.

[Pn1] A Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", Proceedings of Current Trends in Concurrency, Lecture Notes in Computer Science 224.

[SC86] Sistla, Clarke, "Complexity of propositional temporal logics", J. ACM 32(3): 733-749, 1986.

[Tar55] A. Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications", Pacific Journal of Mathematics, vol. 5, pp. 285-309, 1955.

[VHLMD93] S.T. Vuong, H. Janssen, Y. Lu, C. Mathieson, B. Do, "TESTGEN: An Environment for Test Suite Generation and Selection," Computer Communication, March 1993.

Appendix 1: Estelle.Y Alternating Bit Protocol Specification

```
Specification ABP;

CONST
    junkconst = 0 :     int;

VAR
    Send_seq:           boolean;
    Recv_seq:           boolean;
    Recv_buffer_empty:  boolean;
    Recv_buffer_datum:  boolean;
    Recv_buffer_seq:    boolean;
    Send_buffer_empty:  boolean;
    Send_buffer_datum:  boolean;
    Send_buffer_seq:    boolean;

ISP
    SENDrequest         UAccessPoint;
    RECEIVErequest      UAccessPoint;
    DATAinteraction     NAccessPoint;

OSP
    RECEIVEresponse     UAccessPoint;
    DATAinteraction     NAccessPoint;
    SENDconfirm         UAccessPoint;

PDU
    Junk sent_in NAccessPoint;

TIMER
    rexmit_timer        2;

STATE
    ack_wait, estab;

INITIALIZATION
    TO estab
        BEGIN
        Send_seq := 0;
        Recv_seq := 0;
        Send_buffer_empty := TRUE;
        Recv_buffer_empty := TRUE;
        END;

TRANS                                          { trans1 }
    FROM estab
        TO ack_wait
            WHEN SENDrequest
                OUTPUT DATAinteraction, SENDconfirm
                    BEGIN
                    Send_buffer_empty := FALSE;
                    Send_buffer_datum := SENDrequest.udata;
                    Send_buffer_seq := Send_seq;
                    DATAinteraction.ndata.id := 0;
                    DATAinteraction.ndata.conn := 0;
                    DATAinteraction.ndata.data := Send_buffer_datum;
                    DATAinteraction.ndata.seq := Send_buffer_seq;
                    RESET(rexmit_timer);
                    START(rexmit_timer)
                    END;
```

```
TRANS                                                    { trans 2 }
    FROM ack_wait
        TO ack_wait
            WHEN RECEIVErequest
                PROVIDED not (Recv_buffer_empty = TRUE)
                    OUTPUT RECEIVEresponse
                        BEGIN
                        RECEIVEresponse.udata := Recv_buffer_datum;
                        Recv_buffer_empty := TRUE
                        END;

TRANS                                                    { trans3 }
    FROM estab
        TO estab
            WHEN RECEIVErequest
                PROVIDED not (Recv_buffer_empty = TRUE)
                    OUTPUT RECEIVEresponse
                        BEGIN
                        RECEIVEresponse.udata := Recv_buffer_datum;
                        Recv_buffer_empty := TRUE
                        END;

TRANS                                                    { trans4 }
    FROM ack_wait
        TO ack_wait
            PROVIDED TIMEOUT(rexmit_timer)
                OUTPUT DATAinteraction
                    BEGIN
                    DATAinteraction.ndata.id := 0;
                    DATAinteraction.ndata.conn := 0;
                    DATAinteraction.ndata.data := Send_buffer_datum;
                    DATAinteraction.ndata.seq := Send_buffer_seq;
                    RESET(rexmit_timer);
                    START(rexmit_timer)
                    END;

TRANS                                                    { trans5 }
    FROM ack_wait
        TO estab
            WHEN DATAinteraction
                PROVIDED (DATAinteraction.ndata.id = 1) AND
                         (DATAinteraction.ndata.seq = Send_seq)
                    BEGIN
                    Send_buffer_empty := TRUE;
                    IF (Send_seq = TRUE) THEN
                        Send_seq := FALSE
                    ELSE Send_seq := TRUE;
                    RESET(rexmit_timer)
                    END;

TRANS                                                    { trans6 }
    FROM ack_wait
        TO ack_wait
            WHEN DATAinteraction
                PROVIDED (DATAinteraction.ndata.id = 0) AND
                         (Recv_buffer_empty = TRUE)
                    OUTPUT DATAinteraction
                        BEGIN
                        DATAinteraction.ndata.id := 1;
                        DATAinteraction.ndata.conn := 0;
                        DATAinteraction.ndata.data := 0;
                        DATAinteraction.ndata.seq :=
                            DATAinteraction.ndata.seq;
                        IF DATAinteraction.ndata.seq = Recv_seq THEN
                            BEGIN
                            Recv_buffer_empty := FALSE;
```

```
                            Recv_buffer_datum := DATAinteraction.ndata.data;
                            Recv_buffer_seq := DATAinteraction.ndata.seq;
                            IF Recv_seq = TRUE THEN
                                Recv_seq := FALSE
                            ELSE Recv_seq := TRUE
                            END
                        END;

TRANS                                                   { trans7 }
    FROM estab
        TO estab
            WHEN DATAinteraction
                PROVIDED (DATAinteraction.ndata.id = 0) AND
                         (Recv_buffer_empty = TRUE)
                    OUTPUT DATAinteraction
                        BEGIN
                        DATAinteraction.ndata.id := 1;
                        DATAinteraction.ndata.conn := 0;
                        DATAinteraction.ndata.data := 0;
                        DATAinteraction.ndata.seq :=
                            DATAinteraction.ndata.seq;
                        IF DATAinteraction.ndata.seq = Recv_seq THEN
                            BEGIN
                            Recv_buffer_empty := FALSE;
                            Recv_buffer_datum := DATAinteraction.ndata.data;
                            Recv_buffer_seq := DATAinteraction.ndata.seq;
                            IF Recv_seq = TRUE THEN
                                Recv_seq := FALSE
                            ELSE Recv_seq := TRUE
                            END
                        END;

TRANS                                               { trans8 }
    FROM estab
        TO estab
            WHEN DATAinteraction
                PROVIDED (DATAinteraction.ndata.id = 0) AND
                         (Recv_buffer_empty = FALSE)
                    BEGIN
                    Recv_buffer_empty := Recv_buffer_empty
                    END;

TRANS                                               { trans9 }
    FROM ack_wait
        TO ack_wait
            WHEN DATAinteraction
                PROVIDED (DATAinteraction.ndata.id = 0) AND
                         (Recv_buffer_empty = FALSE)
                    BEGIN
                    Recv_buffer_empty := Recv_buffer_empty
                    END;

END.
```

Appendix 2: ASN.1 Alternating Bit Protocol Specification

```
ABP DEFINITIONS ::=
BEGIN

UAccessPoint ::= CHOICE
        {
        SENDrequest,
        SENDconfirm,
        RECEIVErequest,
        RECEIVEresponse
        }

NAccessPoint ::= CHOICE
        {
        DATAinteraction
        }

WireEndPoint ::= CHOICE
        {
        NETdata
        }

Pdu ::=         CHOICE
                {
                Junk
                }

Junk ::=        SEQUENCE
                {
                dummy INTEGER (0..1)
                }

UDataType ::= INTEGER (0..1)                            -- user data

NDataType ::= SEQUENCE
        {
        id      INTEGER {data(0), ack(1)} (0..1),       -- type of message
        conn    INTEGER (0..1),                         -- conn id of sender
        data    UDataType,                              -- user data
        seq     INTEGER (0..1)                          -- sequence number
        }

SENDrequest ::= SEQUENCE
        {
        udata   UDataType
        }

SENDconfirm ::= SEQUENCE
        {
        dummy   INTEGER (0..1)
        }

RECEIVErequest ::= SEQUENCE
        {
        dummy   INTEGER (0..1)
        }

RECEIVEresponse ::= SEQUENCE
        {
        udata   UDataType
```

```
        }

DATAinteraction ::= SEQUENCE
        {
        ndata    NDataType
        }

NETdata ::= SEQUENCE
        {
        ndata    NDataType
        }

END
```

Appendix 3: Ever code for Alternating Bit Protocol

```
-- Specification id = ABP
-- yyparse() = 0

-- Constants: junkconst
-- Variables:
deftype abp7_localvars (record
        Send_seq (bits 1)
        Recv_seq (bits 1)
        Recv_buffer_empty (bits 1)
        Recv_buffer_datum (bits 1)
        Recv_buffer_seq (bits 1)
        Send_buffer_empty (bits 1)
        Send_buffer_datum (bits 1)
        Send_buffer_seq (bits 1));

-- ISPs: SENDrequest RECEIVErequest DATAinteraction

-- OSPs: RECEIVEresponse DATAinteraction SENDconfirm

-- PDUs: Junk

-- Timers: rexmit_timer
deftype abp7_timers_type (record
        rexmit_timer (record running (bits 1) counter (bits 2)));



-- States: ack_wait estab
deftype abp7_mainstate (bits 1 "ack_wait" "estab");

-- PCOs:
deftype UAccessPoint_inqueue (record
        kind (bits 1)
        RECEIVErequest  (record dummy (bits 1))
        SENDrequest  (record udata (bits 1)));

deftype UAccessPoint_outqueue (record
        kind (bits 1)
        SENDconfirm  (record dummy (bits 1))
        RECEIVEresponse  (record udata (bits 1)));

deftype NAccessPoint_queue (record
        DATAinteraction  (record ndata (record id (bits 1) conn (bits 1)
                                 data (bits 1) seq (bits 1))));

-- Free variable for clock ticking
deffreevar clock_tick (bits 1);

-- Ever variables for interaction queues
defvar UAccessPoint_in (array 0 1 (record base (bits 0) queued (bits 1)
                                        content (array 0 0
AccessPoint_inqueue)));
defvar UAccessPoint_out (array 0 1
                           (record base (bits 0) queued (bits 1)
                                   content (array 0 0
AccessPoint_outqueue)));
defvar NAccessPoint (array 0 1
                      (record base (bits 0) queued (bits 1)
                              content (array 0 0 NAccessPoint_queue)));
```

```
-- Ever variables for module abp7 states
defvar abp7 (array 0 1 (record state abp7_mainstate
                               vars abp7_localvars
                               timers abp7_timers_type));


-- Ever predicates for module abp7
defpred abp7_Ctrans1 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 1)
    (not (eq UAccessPoint_in[i0].queued^c 0))
    (eq  UAccessPoint_in[i0].content[UAccessPoint_in[i0].base^c].kind^c 0)
    (lt NAccessPoint[o1].queued^c 1)
    (lt UAccessPoint_out[o0].queued^c 1));

defpred abp7_Atrans1 (n i0 o0 i1 o1)
        (compose
        (becomes abp7[n].vars.Send_buffer_empty^n 0)
        (becomes abp7[n].vars.Send_buffer_datum^n
               UAccessPoint_in[i0].content[UAccessPoint_in[i0].base^c].
               SENDrequest.udata^c)
        (becomes abp7[n].vars.Send_buffer_seq^n abp7[n].vars.Send_seq^c)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
               NAccessPoint[o1].queued^c)].DATAinteraction.ndata.id^n 0)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
               NAccessPoint[o1].queued^c)].DATAinteraction.ndata.conn^n 0)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
               NAccessPoint[o1].queued^c)].DATAinteraction.ndata.data^n
               abp7[n].vars.Send_buffer_datum^c)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
               NAccessPoint[o1].queued^c)].DATAinteraction.ndata.seq^n
               abp7[n].vars.Send_buffer_seq^c)
        (becomes abp7[n].timers.rexmit_timer^n 0)
        (becomes abp7[n].timers.rexmit_timer.running^n 1)
        (becomes UAccessPoint_in[i0].base^n (add UAccessPoint_in[i0].base^c
1))
        (becomes UAccessPoint_in[i0].queued^n (sub
UAccessPoint_in[i0].queued^c 1))
        (becomes NAccessPoint[o1].queued^n (add NAccessPoint[o1].queued^c
1))
        (becomes UAccessPoint_out[o0].content[(add
             UAccessPoint_out[o0].base^c
             UAccessPoint_out[o0].queued^c)].kind^n 1)
        (becomes UAccessPoint_out[o0].queued^n (add
UAccessPoint_out[o0].queued^c 1))
        (becomes abp7[n].state^n 0));

defpred abp7_Ctrans2 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 0)
    (not (eq UAccessPoint_in[i0].queued^c 0))
    (eq  UAccessPoint_in[i0].content[UAccessPoint_in[i0].base^c].kind^c 1)
    (not (eq  abp7[n].vars.Recv_buffer_empty^c 1))
    (lt UAccessPoint_out[o0].queued^c 1));

defpred abp7_Atrans2 (n i0 o0 i1 o1)
        (compose
        (becomes UAccessPoint_out[o0].content[(add
             UAccessPoint_out[o0].base^c
             UAccessPoint_out[o0].queued^c)].RECEIVEresponse.udata^n
             abp7[n].vars.Recv_buffer_datum^c)
        (becomes abp7[n].vars.Recv_buffer_empty^n 1)
        (becomes UAccessPoint_in[i0].base^n (add UAccessPoint_in[i0].base^c
1))
        (becomes UAccessPoint_in[i0].queued^n (sub
UAccessPoint_in[i0].queued^c 1))
```

```
        (becomes UAccessPoint_out[o0].content[(add
             UAccessPoint_out[o0].base^c
             UAccessPoint_out[o0].queued^c)].kind^n 0)
        (becomes UAccessPoint_out[o0].queued^n (add
UAccessPoint_out[o0].queued^c 1))
        (becomes abp7[n].state^n 0));

defpred abp7_Ctrans3 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 1)
    (not (eq UAccessPoint_in[i0].queued^c 0))
    (eq  UAccessPoint_in[i0].content[UAccessPoint_in[i0].base^c].kind^c 1)
    (not (eq  abp7[n].vars.Recv_buffer_empty^c 1))
    (lt UAccessPoint_out[o0].queued^c 1));

defpred abp7_Atrans3 (n i0 o0 i1 o1)
        (compose
        (becomes UAccessPoint_out[o0].content[(add
               UAccessPoint_out[o0].base^c
               UAccessPoint_out[o0].queued^c)].RECEIVEresponse.udata^n
               abp7[n].vars.Recv_buffer_datum^c)
        (becomes abp7[n].vars.Recv_buffer_empty^n 1)
        (becomes UAccessPoint_in[i0].base^n (add UAccessPoint_in[i0].base^c
1))
        (becomes UAccessPoint_in[i0].queued^n (sub
UAccessPoint_in[i0].queued^c 1))
        (becomes UAccessPoint_out[o0].content[(add
               UAccessPoint_out[o0].base^c
               UAccessPoint_out[o0].queued^c)].kind^n 0)
        (becomes UAccessPoint_out[o0].queued^n (add
UAccessPoint_out[o0].queued^c 1))
        (becomes abp7[n].state^n 1));

defpred abp7_Ctrans4 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 0)
    (ge abp7[n].timers.rexmit_timer.counter^c 2)
    (lt NAccessPoint[o1].queued^c 1));

defpred abp7_Atrans4 (n i0 o0 i1 o1)
        (compose
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.id^n 0)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.conn^n 0)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.data^n
             abp7[n].vars.Send_buffer_datum^c)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.seq^n
             abp7[n].vars.Send_buffer_seq^c)
        (becomes abp7[n].timers.rexmit_timer^n 0)
        (becomes abp7[n].timers.rexmit_timer.running^n 1)
        (becomes NAccessPoint[o1].queued^n (add NAccessPoint[o1].queued^c
1))
        (becomes abp7[n].state^n 0));

defpred abp7_Ctrans5 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 0)
    (not (eq NAccessPoint[i1].queued^c 0))
    (and (eq  NAccessPoint[i1].content[NAccessPoint[i1].base^c].
             DATAinteraction.ndata.id^c 1)
         (eq NAccessPoint[i1].content[NAccessPoint[i1].base^c].
             DATAinteraction.ndata.seq^c abp7[n].vars.Send_seq^c)));

defpred abp7_Atrans5 (n i0 o0 i1 o1)
        (compose
        (becomes abp7[n].vars.Send_buffer_empty^n 1)
        (if (eq  abp7[n].vars.Send_seq^c 1)
```

```
        (becomes abp7[n].vars.Send_seq^n 0)
        (becomes abp7[n].vars.Send_seq^n 1))
        (becomes abp7[n].timers.rexmit_timer^n 0)
        (becomes NAccessPoint[i1].base^n (add NAccessPoint[i1].base^c 1))
        (becomes NAccessPoint[i1].queued^n (sub NAccessPoint[i1].queued^c
1))
        (becomes abp7[n].state^n 1));

defpred abp7_Ctrans6 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 0)
    (not (eq NAccessPoint[i1].queued^c 0))
    (and (eq  NAccessPoint[i1].content[NAccessPoint[i1].base^c].
              DATAinteraction.ndata.id^c 0)
         (eq  abp7[n].vars.Recv_buffer_empty^c 1))
    (lt NAccessPoint[o1].queued^c 1));

defpred abp7_Atrans6 (n i0 o0 i1 o1)
        (compose
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
              NAccessPoint[o1].queued^c)].DATAinteraction.ndata.id^n 1)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
              NAccessPoint[o1].queued^c)].DATAinteraction.ndata.conn^n 0)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
              NAccessPoint[o1].queued^c)].DATAinteraction.ndata.data^n 0)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
                                           NAccessPoint[o1].queued^c)].
                                      DATAinteraction.ndata.seq^n
              NAccessPoint[i1].content[NAccessPoint[i1].base^c].
                DATAinteraction.ndata.seq^c)
        (if (eq NAccessPoint[i1].content[NAccessPoint[i1].base^c].
                DATAinteraction.ndata.seq^c abp7[n].vars.Recv_seq^c)
        (compose
        (becomes abp7[n].vars.Recv_buffer_empty^n 0)
        (becomes abp7[n].vars.Recv_buffer_datum^n
              NAccessPoint[i1].content[NAccessPoint[i1].base^c].
                DATAinteraction.ndata.data^c)
        (becomes abp7[n].vars.Recv_buffer_seq^n
              NAccessPoint[i1].content[NAccessPoint[i1].base^c].
              DATAinteraction.ndata.seq^c)
        (if (eq  abp7[n].vars.Recv_seq^c 1)
            (becomes abp7[n].vars.Recv_seq^n 0)
            (becomes abp7[n].vars.Recv_seq^n 1)))
        (becomes abp7^n abp7^c))
        (becomes NAccessPoint[i1].base^n (add NAccessPoint[i1].base^c 1))
        (becomes NAccessPoint[i1].queued^n (sub NAccessPoint[i1].queued^c
1))
        (becomes NAccessPoint[o1].queued^n (add NAccessPoint[o1].queued^c
1))
        (becomes abp7[n].state^n 0));

defpred abp7_Ctrans7 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 1)
    (not (eq NAccessPoint[i1].queued^c 0))
    (and (eq
NAccessPoint[i1].content[NAccessPoint[i1].base^c].DATAinteraction.ndata.id^c
0)
         (eq  abp7[n].vars.Recv_buffer_empty^c 1))
    (lt NAccessPoint[o1].queued^c 1));

defpred abp7_Atrans7 (n i0 o0 i1 o1)
        (compose
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.id^n 1)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.conn^n 0)
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.data^n 0)
```

```
        (becomes NAccessPoint[o1].content[(add NAccessPoint[o1].base^c
             NAccessPoint[o1].queued^c)].DATAinteraction.ndata.seq^n
             NAccessPoint[i1].content[NAccessPoint[i1].base^c].
               DATAinteraction.ndata.seq^c)
        (if (eq  NAccessPoint[i1].content[NAccessPoint[i1].base^c].
                   DATAinteraction.ndata.seq^c
                 abp7[n].vars.Recv_seq^c)
        (compose
        (becomes abp7[n].vars.Recv_buffer_empty^n 0)
        (becomes abp7[n].vars.Recv_buffer_datum^n
             NAccessPoint[i1].content[NAccessPoint[i1].base^c].
               DATAinteraction.ndata.data^c)
        (becomes abp7[n].vars.Recv_buffer_seq^n
             NAccessPoint[i1].content[NAccessPoint[i1].base^c].
               DATAinteraction.ndata.seq^c)
        (if (eq  abp7[n].vars.Recv_seq^c 1)
        (becomes abp7[n].vars.Recv_seq^n 0)
        (becomes abp7[n].vars.Recv_seq^n 1)))
        (becomes abp7^n abp7^c))
        (becomes NAccessPoint[i1].base^n (add NAccessPoint[i1].base^c 1))
        (becomes NAccessPoint[i1].queued^n (sub NAccessPoint[i1].queued^c
1))
        (becomes NAccessPoint[o1].queued^n (add NAccessPoint[o1].queued^c
1))
        (becomes abp7[n].state^n 1));

defpred abp7_Ctrans8 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 1)
    (not (eq NAccessPoint[i1].queued^c 0))
    (and (eq  NAccessPoint[i1].content[NAccessPoint[i1].base^c].
                DATAinteraction.ndata.id^c 0)
         (eq  abp7[n].vars.Recv_buffer_empty^c 0)));

defpred abp7_Atrans8 (n i0 o0 i1 o1)
        (compose
        (becomes abp7[n].vars.Recv_buffer_empty^n
abp7[n].vars.Recv_buffer_empty^c)
        (becomes NAccessPoint[i1].base^n (add NAccessPoint[i1].base^c 1))
        (becomes NAccessPoint[i1].queued^n (sub NAccessPoint[i1].queued^c
1))
        (becomes abp7[n].state^n 1));

defpred abp7_Ctrans9 (n i0 o0 i1 o1) (and
    (eq  abp7[n].state^c 0)
    (not (eq NAccessPoint[i1].queued^c 0))
    (and (eq  NAccessPoint[i1].content[NAccessPoint[i1].base^c].
                DATAinteraction.ndata.id^c 0)
         (eq  abp7[n].vars.Recv_buffer_empty^c 0)));

defpred abp7_Atrans9 (n i0 o0 i1 o1)
        (compose
        (becomes abp7[n].vars.Recv_buffer_empty^n
abp7[n].vars.Recv_buffer_empty^c)
        (becomes NAccessPoint[i1].base^n (add NAccessPoint[i1].base^c 1))
        (becomes NAccessPoint[i1].queued^n (sub NAccessPoint[i1].queued^c
1))
        (becomes abp7[n].state^n 0));

defpred abp7_timertick (n) (compose
    (if (and (eq abp7[n].timers.rexmit_timer.running^c 1)
             (lt abp7[n].timers.rexmit_timer.counter^c 2))
        (becomes abp7[n].timers.rexmit_timer.counter^n
                 (add abp7[n].timers.rexmit_timer.counter^c 1))
        (becomes abp7[n].timers^n abp7[n].timers^c)));

defpred abp7_nextstate (n i0 o0 i1 o1) (or
    (if (eq clock_tick^c 0) (or
```

```
        (if (abp7_Ctrans1 n i0 o0 i1 o1) (abp7_Atrans1 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans2 n i0 o0 i1 o1) (abp7_Atrans2 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans3 n i0 o0 i1 o1) (abp7_Atrans3 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans4 n i0 o0 i1 o1) (abp7_Atrans4 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans5 n i0 o0 i1 o1) (abp7_Atrans5 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans6 n i0 o0 i1 o1) (abp7_Atrans6 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans7 n i0 o0 i1 o1) (abp7_Atrans7 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans8 n i0 o0 i1 o1) (abp7_Atrans8 n i0 o0 i1 o1) FALSE)
        (if (abp7_Ctrans9 n i0 o0 i1 o1) (abp7_Atrans9 n i0 o0 i1 o1) FALSE)
        (if (and
                (not (abp7_Ctrans1 n i0 o0 i1 o1))
                (not (abp7_Ctrans2 n i0 o0 i1 o1))
                (not (abp7_Ctrans3 n i0 o0 i1 o1))
                (not (abp7_Ctrans4 n i0 o0 i1 o1))
                (not (abp7_Ctrans5 n i0 o0 i1 o1))
                (not (abp7_Ctrans6 n i0 o0 i1 o1))
                (not (abp7_Ctrans7 n i0 o0 i1 o1))
                (not (abp7_Ctrans8 n i0 o0 i1 o1))
                (not (abp7_Ctrans9 n i0 o0 i1 o1)))
            (becomes abp7^n abp7^c) FALSE)) FALSE)
    (if (eq clock_tick^c 1) (abp7_timertick n) FALSE));

defpred abp7_init (n o0 o1) (and
    (eq abp7[n].state^c 1)
          (eq NAccessPoint[o1].base^c 0)
          (eq NAccessPoint[o1].queued^c 0)
          (eq UAccessPoint_out[o0].base^c 0)
          (eq UAccessPoint_out[o0].queued^c 0)
    (eq abp7[n].vars.Send_seq^c 0)
    (eq abp7[n].vars.Recv_seq^c 0)
    (eq abp7[n].vars.Send_buffer_empty^c 1)
    (eq abp7[n].vars.Recv_buffer_empty^c 1)
    (eq abp7[n].timers^c 0));

defprop init (and
    (abp7_init 0 0 0)
    (abp7_init 1 1 1));

defprop nextstate (or
    (abp7_nextstate 0 0 0 0 1)
    (abp7_nextstate 1 1 1 1 0));
```

Appendix 4: Ever command file for verification

```
-- Ever source file for specifying behaviour of sender
-- and receiver modules for alternating bit protocol.

-- same as original except all universal quantifiers
-- changed to existential  (01Nov1993)

-- In definitions of receive_datum_x, changed original
-- AX (intended for within scope of a single module) from
-- (x /\ AXx') to (x /\ AXA[x U x']).  (04Nov1993)

-- Want to test send_datum_x and receive_datum_x definitions
-- with universal operators in them to see if they are
-- totally false.   (05Nov1993)

-- Modified for case with unreliable underlying link
-- and implementation of timers in abp specification.
-- (Tue 16Nov1993)

-- Added action code for data loss in unreliable link. (Tue 16Nov1993)

-- Added 2 fairness constraints to specify that
-- unreliable link regularily delivers data reliably. (Tue 16Nov1993)

-- Added 2 fairness constraints for timer implementation. (Tue 16Nov1993)


-- Define two unconstrained variables for user data
deffreevar user_txdatum (bits 1);
deffreevar user_rxdatum (bits 1);

-- Define nextstate relation of user modules
defprop user_next (or
    -- SendRequest (send one whenever possible)
    (if (eq UAccessPoint_in[0].queued^c 0)
        (compose
            (becomes UAccessPoint_in[0].content[0].SENDrequest.udata^n
                     user_txdatum^c)
            (becomes UAccessPoint_in[0].content[0].kind^n 0)
            (becomes UAccessPoint_in[0].queued^n 1))
        FALSE)
    -- SendConfirm (dequeue SendConfirm whenever one is delivered)
    (if (and (gt UAccessPoint_out[0].queued^c 0)
             (eq UAccessPoint_out[0].content[0].kind^c 1))
        (becomes UAccessPoint_out[0].queued^n 0)
        FALSE)
    -- ReceiveRequest (send one whenever possible)
    (if (eq UAccessPoint_in[1].queued^c 0)
        (compose
            (becomes UAccessPoint_in[1].content[0].kind^n 1)
            (becomes UAccessPoint_in[1].queued^n 1))
        FALSE)
    -- ReceiveResponse (dequeue one whenever one arrives)
    (if (and (gt UAccessPoint_out[1].queued^c 0)
             (eq UAccessPoint_out[1].content[0].kind^c 0))
        (compose
            (becomes user_rxdatum^n
                     UAccessPoint_out[1].content[0].RECEIVEresponse.udata^c)
            (becomes UAccessPoint_out[1].queued^n 0))
        FALSE)
```

```
    -- Otherwise (always allow a no-op step without confining free
variables)
    (becomes UAccessPoint_in[0].queued^n UAccessPoint_in[0].queued^c)
    );


-- Define nextstate relation of unreliable link (Tue16Nov1993)
defprop unreliable_link_next (or
    (if (gt NAccessPoint[0].queued^c 0)
        (becomes NAccessPoint[0].queued^n 0) FALSE)
    (if (gt NAccessPoint[1].queued^c 0)
        (becomes NAccessPoint[1].queued^n 0) FALSE)
    (becomes NAccessPoint^n NAccessPoint^c));



-- Fairness constraints
--
-- 1.   sender sends SENDrequest requests infinitely often.
-- 2.   sender dequeues SENDconfirm responses infinitely often.*
-- 3.   receiver sends RECEIVErequest requests infinitely often.
-- 4.   receiver dequeues RECEIVEresponse responses infinitely often.*

-- * Maybe should make these happen immediately in user_next nextstate
--   relation.

-- Should add fairness constraints for 0 datum sent infinitely often
-- and 1 datum sent infinitely often.  Probably best to restrict
-- fairness constraint #1 below to datum 0 and add a new similar
-- fairness constraint for datum 1.     Fr 29Oct1993
-- See fair1a_basic and fair1b_basic below      Sun 31Oct1993

defprop fair1_basic (and
    (eq UAccessPoint_in[0].queued^c 1)
    (eq UAccessPoint_in[0].content[0].kind^c 0));

defprop fair1a_basic (and
    (eq UAccessPoint_in[0].queued^c 1)
    (eq UAccessPoint_in[0].content[0].kind^c 0)
    (eq UAccessPoint_in[0].content[0].SENDrequest.udata^c 0));

defprop fair1b_basic (and
    (eq UAccessPoint_in[0].queued^c 1)
    (eq UAccessPoint_in[0].content[0].kind^c 0)
    (eq UAccessPoint_in[0].content[0].SENDrequest.udata^c 1));


defprop fair2_basic (eq UAccessPoint_out[0].queued^c 0);

defprop fair3_basic (and
    (eq UAccessPoint_in[1].queued^c 1)
    (eq UAccessPoint_in[1].content[0].kind^c 1));

defprop fair4_basic (eq UAccessPoint_out[1].queued^c 0);


-- Additional fairness constraints for unreliable link (Tue16Nov1993)
defprop unreliable_fair1 (eq NAccessPoint[0].queued^c 1);
defprop unreliable_fair2 (eq NAccessPoint[1].queued^c 1);

-- Additional fairness constraints for time implementation (Tue16Nov1993)
defprop timer_fair1 (eq clock_tick^c 0);
defprop timer_fair2 (eq clock_tick^c 1);


-- Commands for evaluating (EG TRUE) under basic fairness constraints
```

```
-- defprop global_nextstate (or (nextstate) (user_next)); -- version of
31oct

-- Version of 01nov that include a "keep stable" statement in
-- global disjunction of nextstate relation.
defprop global_nextstate (or (nextstate) (user_next)
                             (unreliable_link_next));

setdefaultnextstate (global_nextstate)
                    (fair1a_basic) (fair1b_basic)
                    (fair2_basic) (fair3_basic) (fair4_basic)
                    (unreliable_fair1) (unreliable_fair2)
                    (timer_fair1) (timer_fair2);

defprop global_init (and (init) (eq UAccessPoint_in^c 0));


-- Output [1] on Tue 16 Nov 1993.
-- defprop egtrue (EG TRUE);
-- printstring "EG TRUE";
-- printprop (egtrue);
-- printsize (egtrue);

-- First, define a set of events in send and receive users,
-- and ABP protocol modules.

-- true when a SENDrequest has been submitted by the sending
-- user and is awaiting processing by the ABP send module and
-- the datum value of this request is x.
defpred usend (x) (and
    (eq UAccessPoint_in[0].queued^c 1)
    (eq UAccessPoint_in[0].content[0].kind^c 0)
    (eq UAccessPoint_in[0].content[0].SENDrequest.udata^c x));

-- true when the sending user has no send request pending servicing
-- by ABP sender
defprop usendnothing (eq UAccessPoint_in[0].queued^c 0);

-- true when a SENDconfirm has been submitted by
-- the ABP sender module and is awaiting processing by the
-- sending user.
defprop usendconfirm (and
    (eq UAccessPoint_out[0].queued^c 1)
    (eq UAccessPoint_out[0].content[0].kind^c 1));

-- true when data packet containing datum value x and sequence
-- number seq has been submitted for transmission by the
-- ABP sender module and is awaiting transfer over the
-- underlying link.
defpred asend (x seq) (and
    (eq NAccessPoint[1].queued^c 1)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.id^c 0)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.conn^c 0)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.data^c x)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.seq^c seq));

-- true when ABP sender module has no request to transmit a packet
-- over underlying link pending.
-- N.B. This is valid for reliable underlying link and for
--      unreliable link which can only lose data but not duplicate it.
defprop asendnothing (eq NAccessPoint[1].queued^c 0);

-- true when a data packet containing datum value x and
-- sequence number seq is delivered to the ABP receiver module by the
-- underlying link and has not yet been processed by
-- the ABP receiver module.
-- (in model of underlying unreliable link supporting only
```

```
-- data loss, this predicate is not accurate because data can
-- be delivered and subsequently removed by actions of
-- underlying unreliable link before ABP receiver processes
-- delivered data)
-- (would be accurate with model of unreliable underlying link
-- with 2 extra queues between pair of ABP modules)
defpred arcv (x seq) (and
    (eq NAccessPoint[1].queued^c 1)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.id^c 0)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.conn^c 0)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.data^c x)
    (eq NAccessPoint[1].content[0].DATAinteraction.ndata.seq^c seq));

-- true when there is no packet pending receipt by the ABP receiver
-- module in its queue from the underlying link.
defprop arcvnothing (eq NAccessPoint[1].queued^c 0);

-- true when the ABP receiver module has submitted a request to transmit
-- an acknowledgement packet with sequence number seq.
-- (not accurate in model of unreliable link with only data loss).
defpred aacksend (seq) (and
    (eq NAccessPoint[0].queued^c 1)
    (eq NAccessPoint[0].content[0].DATAinteraction.ndata.id^c 1)
    (eq NAccessPoint[0].content[0].DATAinteraction.ndata.conn^c 0)
    (eq NAccessPoint[0].content[0].DATAinteraction.ndata.seq^c seq));

-- true when an acknowledgement packet containing sequence number
-- seq has been delivered to the ABP sender module by the underlying link.
-- (not accurate in model of unreliable link with only data loss).
defpred aackrcv (seq) (and
    (eq NAccessPoint[0].queued^c 1)
    (eq NAccessPoint[0].content[0].DATAinteraction.ndata.id^c 1)
    (eq NAccessPoint[0].content[0].DATAinteraction.ndata.conn^c 0)
    (eq NAccessPoint[0].content[0].DATAinteraction.ndata.seq^c seq));

-- true when a RECEIVEresponse containing value x has been submitted
-- by the ABP receiver module and has not been processed by the
-- receiving user.
defpred urcv (x) (and
    (eq UAccessPoint_out[1].queued^c 1)
    (eq UAccessPoint_out[1].content[0].RECEIVEresponse.udata^c x)
    (eq UAccessPoint_out[1].content[0].kind^c 0));

-- true when there is no RECEIVEresponse pending servicing by receiving
-- user in queue from ABP receiver.
defprop urcvnothing (eq UAccessPoint_out[1].queued^c 0);

-- true when a RECEIVErequest has been submitted by the receiving
-- user and is awaiting processing by the ABP receive module.
defprop urcvreq (and
    (eq UAccessPoint_in[1].queued^c 1)
    (eq UAccessPoint_in[1].content[0].kind^c 1));


-- Properties to verify
-- All properites p are to be evaluated in context of
-- (implies (global_init) (p));

-- Assert that no data requests are sent by ABP sender module
-- while it is in the ESTAB state.
-- AG( (state=ESTAB) implies ASendNothing) )
defprop ag_state_eq_ESTAB_implies_ASendNothing
    (AG (implies (eq abp6[0].state^c 1) (asendnothing)));

defprop init_implies_ag_state_eq_ESTAB_implies_ASendNothing
    (implies (global_init)
             (ag_state_eq_ESTAB_implies_ASendNothing));
```

```
printstring "init_implies_ag_state_eq_ESTAB_implies_ASendNothing";
printprop (init_implies_ag_state_eq_ESTAB_implies_ASendNothing);
printsize (init_implies_ag_state_eq_ESTAB_implies_ASendNothing);


-- Assert that Send_buffer_empty true iff main state is ESTAB.
defprop ag_send_buffer_empty_equiv_estab_state
    (AG (equiv (eq abp6[0].vars.Send_buffer_empty^c 1)
               (eq abp6[0].state^c 1)));

-- printstring "ag_send_buffer_empty_equiv_estab_state";
-- printprop (ag_send_buffer_empty_equiv_estab_state);
-- printsize (ag_send_buffer_empty_equiv_estab_state);

defprop init_implies_ag_send_buffer_empty_equiv_estab_state
    (implies (global_init)
             (ag_send_buffer_empty_equiv_estab_state));

printstring "init_implies_ag_send_buffer_empty_equiv_estab_state";
printprop (init_implies_ag_send_buffer_empty_equiv_estab_state);
printsize (init_implies_ag_send_buffer_empty_equiv_estab_state);


-- Assert that whenever sendbuffer is not empty that Send_buffer_seq
-- variable equals Send_seq variable.
-- AG( not(Send_buffer_empty) implies (Send_buffer_seq = Send_seq) )
defprop ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq
    (AG (implies (not (eq abp6[0].vars.Send_buffer_empty^c 1))
                 (eq abp6[0].vars.Send_buffer_seq^c
                     abp6[0].vars.Send_seq^c)));

defprop init_ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq
    (implies (global_init)

(ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq));

printstring
"init_ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq";
printprop
(init_ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq);
printsize
(init_ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq);


-- Assert that the ABP sender module is always in the ESTAB (1) state
-- or the ACK_WAIT (0) state.
defprop ag_state_estab_or_ackwait
    (AG (or (eq abp6[0].state^c 0)
            (eq abp6[0].state^c 1)));

defprop init_ag_state_estab_or_ackwait
    (implies (global_init)
             (ag_state_estab_or_ackwait));

printstring "init_ag_state_estab_or_ackwait";
printprop (init_ag_state_estab_or_ackwait);
printsize (init_ag_state_estab_or_ackwait);

-- Assert that the ABP sender module never stays in
-- either state permanently.
-- AG( AF(state=ESTAB) and AF(state=ACK_WAIT) )
defprop ag_af_state_ESTAB_and_af_state_ACKWAIT
    (AG (and (AF (eq abp6[0].state^c 0))
             (AF (eq abp6[0].state^c 1))));

defprop init_ag_af_state_ESTAB_and_af_state_ACKWAIT
```

```
    (implies (global_init)
             (ag_af_state_ESTAB_and_af_state_ACKWAIT));

printstring "init_ag_af_state_ESTAB_and_af_state_ACKWAIT";
printprop (init_ag_af_state_ESTAB_and_af_state_ACKWAIT);
printsize (init_ag_af_state_ESTAB_and_af_state_ACKWAIT);


-- Assert that only one request is processed for each ESTAB->ACK_WAIT->ESTAB
-- cycle in the ABP sender module.
-- (formula f1)
defpred one_send_request_at_a_time (x)
    (AG (implies (and (usend x) (eq abp6[0].state^c 0))
                 (AU (usend x)
                     (and (usend x) (eq abp6[0].state^c 1))))));

defprop forall_x_one_send_request_at_a_time
    (and (one_send_request_at_a_time 0)
         (one_send_request_at_a_time 1));

defprop init_forall_x_one_send_request_at_a_time
    (implies (global_init) (forall_x_one_send_request_at_a_time));

printstring "init_forall_x_one_send_request_at_a_time";
printprop (init_forall_x_one_send_request_at_a_time);
printsize (init_forall_x_one_send_request_at_a_time);


-- Assert that a packet will be repeatedly transmitted until an
-- acknowledgement is received when a send request is in the
-- queue from the user when in the ESTAB state.
defpred repeat_transmit_packet (x)
    (AG (implies (and (eq abp6[0].state^c 1) (usend x))
                 (AU (and (asendnothing) (eq abp6[0].state^c 1))
                     (AU (and (or (asendnothing)
                                  (asend x abp6[0].vars.Send_seq^c))
                              (eq abp6[0].state^c 0))
                         (eq abp6[0].state^c 1))))));

defprop forall_x_repeat_transmit_packet
    (and (repeat_transmit_packet 0)
         (repeat_transmit_packet 1));

defprop init_forall_x_repeat_transmit_packet
    (implies (global_init)
             (forall_x_repeat_transmit_packet));


printstring "init_forall_x_repeat_transmit_packet";
printprop (init_forall_x_repeat_transmit_packet);
printsize (init_forall_x_repeat_transmit_packet);


-- (Mon 06Dec1993)
-- Ever command file for rerunning verification
-- on corrections to formulae that did evaluate to
-- tautologies in original test (run_29nov).

-- Assert that when an acknowledgement is received for current sequence
-- number while in ACK_WAIT state, the state will be eventually
-- returned to ESTAB and the current sequence number will be incremented.
-- (formula f3)
defpred new_ack_rcvd_leads_to_incr_seq_num_and_estab (x s)
    (AG (implies
          (and (eq abp6[0].state^c 0)
               (eq abp6[0].vars.Send_seq^c s)
               (aackrcv s)
```

```
                (eq abp6[0].vars.Send_buffer_datum^c x))
           (AF (and (eq abp6[0].state^c 1)
                    (eq abp6[0].vars.Send_seq^c (add s 1))))));

defprop new_forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab
    (and (new_ack_rcvd_leads_to_incr_seq_num_and_estab 0 0)
         (new_ack_rcvd_leads_to_incr_seq_num_and_estab 0 1)
         (new_ack_rcvd_leads_to_incr_seq_num_and_estab 1 0)
         (new_ack_rcvd_leads_to_incr_seq_num_and_estab 1 1));

defprop init_new_forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab
    (implies (global_init)
             (new_forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab));

printstring "init_new_forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab";
printprop (init_new_forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab);
printsize (init_new_forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab);


-- Assert that only following events occur after a send request
-- is received by the ABP sender module.
--  While still in ESTAB state, only can receive acks from previous
--  request.
--  Eventually will change to ACK_WAIT state.
--  While in ACK_WAIT state, can send retransmissions of data packet,
--  ABP receiver module can receive packet, ABP receiver module
--  can send acknowledgement packets and ABP sender module
--  can receive acknowledgement packets.
--  Eventually will return to ESTAB state with Send_seq variable
--  incremented.
-- (formula f4)
defpred exists_y_asend_y_s_minus_1 (s)
    (and (eq NAccessPoint[1].queued^c 1)
         (eq NAccessPoint[1].content[0].DATAinteraction.ndata.id^c 0)
         (eq NAccessPoint[1].content[0].DATAinteraction.ndata.conn^c 0)
         (eq NAccessPoint[1].content[0].DATAinteraction.ndata.seq^c
             (sub s 1)));

defpred new_send_cycle (x s)
    (AG (implies
            (and (usend x) (eq abp6[0].state^c 1)
                 (eq abp6[0].vars.Send_seq^c s))
            (AU (and (eq abp6[0].state^c 1)
                     (or (asendnothing)
                         (aackrcv (sub s 1))
                         (exists_y_asend_y_s_minus_1 s)))
                (AU (and (eq abp6[0].state^c 0)
                         (or (asend x s) (asendnothing)
                             (arcvnothing) (arcv x s)
                             (aacksend s) (aackrcv s)))
                    (and (eq abp6[0].state^c 1)
                         (eq abp6[0].vars.Send_seq^c (add s 1))))))));

defprop new_forall_x_s_send_cycle
    (and (new_send_cycle 0 0) (new_send_cycle 0 1)
         (new_send_cycle 1 0) (new_send_cycle 1 1));

defprop init_new_forall_x_s_send_cycle
    (implies (global_init) (new_forall_x_s_send_cycle));

printstring "init_new_forall_x_s_send_cycle";
printprop (init_new_forall_x_s_send_cycle);
printsize (init_new_forall_x_s_send_cycle);

(( test run on 09 dec 1993 ))
-- Ever command file for formulae to show that exactly
-- one send data request from sending user matches
```

```
-- exactly one receive data request from the receiving user.

-- Abbreviations
defprop sendbufferempty (eq abp7[0].vars.Send_buffer_empty^c 1);
defprop recvbufferempty (eq abp7[1].vars.Recv_buffer_empty^c 1);

-- First show one exists x.USend(x) per time Send_buffer filled
defprop usend_matches_send_buffer_fill_nest5
    (AU (and (not (usendnothing)) (sendbufferempty))
        (and (usendnothing) (not (sendbufferempty))));

defprop usend_matches_send_buffer_fill_nest4
    (AU (not (xor (usendnothing) (sendbufferempty))) -- optional interval
        (and (not (usendnothing)) (sendbufferempty)
             (usend_matches_send_buffer_fill_nest5)));

defprop usend_matches_send_buffer_fill_nest3
    (AU (and (usendnothing) (not (sendbufferempty)))
        (usend_matches_send_buffer_fill_nest4));

defprop usend_matches_send_buffer_fill_nest2
    (AU (and (not (usendnothing)) (sendbufferempty))
        (and (usendnothing) (not (sendbufferempty))
             (usend_matches_send_buffer_fill_nest3)));

defprop usend_matches_send_buffer_fill_nest1
    (AU (and (not (usendnothing)) (not (sendbufferempty)))  -- optional
interval
        (and (not (usendnothing)) (sendbufferempty)
             (usend_matches_send_buffer_fill_nest2)));

defprop usend_matches_send_buffer_fill
  (AG (implies
        (not (usendnothing))
        (usend_matches_send_buffer_fill_nest1)));

defprop init_usend_matches_send_buffer_fill
  (implies (global_init) (usend_matches_send_buffer_fill));

printstring "init_usend_matches_send_buffer_fill";
printprop (init_usend_matches_send_buffer_fill);
printsize (init_usend_matches_send_buffer_fill);


-- Second, show one Send_buffer fill event per send sequence number
defpred send_buffer_fill_sequence (s)
 (AU (sendbufferempty)
     (and (not (sendbufferempty)) (eq abp7[0].vars.Send_seq^c s)
          (AU (and (not (sendbufferempty)) (eq abp7[0].vars.Send_seq^c s))
              (and (sendbufferempty) (eq abp7[0].vars.Send_seq^c (add s 1))
                   (AU (and (sendbufferempty) (eq abp7[0].vars.Send_seq^c
(add s 1)))
                       (and (not (sendbufferempty)) (eq
abp7[0].vars.Send_seq^c (add s 1))))))
                      ));


defprop send_buffer_fill_matches_seq_number
    (AG (implies (sendbufferempty)
                 (or (send_buffer_fill_sequence 0)
                     (send_buffer_fill_sequence 1))));

defprop init_send_buffer_fill_matches_seq_number
    (implies (global_init) (send_buffer_fill_matches_seq_number));

printstring "init_send_buffer_fill_matches_seq_number";
printprop (init_send_buffer_fill_matches_seq_number);
```

```
printsize (init_send_buffer_fill_matches_seq_number);


-- Third, show one send sequence number for each receive sequence
-- number and each time Recv_buffer filled Recv_seq is incremented.
defpred recv_buffer_fill_sequence (s)
 (AU (recvbufferempty)
     (and (not (recvbufferempty))
          (eq abp7[1].vars.Recv_seq^c s)
          (AU (and (not (recvbufferempty)) (eq abp7[1].vars.Recv_seq^c s))
              (and (recvbufferempty)
                   (AU (and (recvbufferempty) (eq abp7[1].vars.Recv_seq^c
s))
                       (and (not (recvbufferempty))
                            (eq abp7[1].vars.Recv_seq^c (add s 1)))))))));

defprop recv_buffer_fill_sequence_0 (recv_buffer_fill_sequence 0);
defprop recv_buffer_fill_sequence_1 (recv_buffer_fill_sequence 1);

defprop receive_buf_filled_matches_seq_no
    (AG (implies (recvbufferempty)
                 (or (recv_buffer_fill_sequence_0)
                     (recv_buffer_fill_sequence_1))));

defprop init_receive_buf_filled_matches_seq_no
    (implies (global_init)
             (receive_buf_filled_matches_seq_no));

printstring "init_receive_buf_filled_matches_seq_no";
printprop (init_receive_buf_filled_matches_seq_no);
printsize (init_receive_buf_filled_matches_seq_no);


-- Fourth, show one Recv_buffer fill event per exists x.URcv(x) event.
defprop urcv_buff_fill_seq_nest5
    (AU (and (urcvnothing) (not (recvbufferempty)))
        (and (not (urcvnothing)) (recvbufferempty)));

defprop urcv_buff_fill_seq_nest4
    (AU (not (xor (urcvnothing) (recvbufferempty))) -- optional interval
        (and (urcvnothing)
             (not (recvbufferempty))
             (urcv_buff_fill_seq_nest5)));

defprop urcv_buff_fill_seq_nest3
    (AU (and (not (urcvnothing)) (recvbufferempty))
        (urcv_buff_fill_seq_nest4));

defprop urcv_buff_fill_seq_nest2
    (AU (and (urcvnothing) (not (recvbufferempty)))
        (and (not (urcvnothing)) (recvbufferempty)
             (urcv_buff_fill_seq_nest3)));

defprop urcv_buff_fill_seq
    (AU (and (urcvnothing) (recvbufferempty))       -- optional interval
        (and (urcvnothing) (not (recvbufferempty))
             (urcv_buff_fill_seq_nest2)));

defprop urcv_matches_recv_buff_fill
    (AG (implies (urcvnothing)
                 (urcv_buff_fill_seq)));

defprop init_urcv_matches_recv_buff_fill
    (implies (global_init)
             (urcv_matches_recv_buff_fill));

printstring "init_urcv_matches_recv_buff_fill";
```

```
printprop (init_urcv_matches_recv_buff_fill);
printsize (init_urcv_matches_recv_buff_fill);

(( test run on 12 dec 1993 ))
-- Ever command file to verify property that data value
-- of a user's send request is preserved from time of
-- user submitting the request to time of the receiving
-- user receiving the data.

-- Interval when data packet has been received by ABP receiver
-- module and receiving user has submitted a receive request
-- until received read the data.
defpred data_preserved_interval7 (x)
  (and (eq abp7[1].vars.Recv_buffer_empty^c 1)
       (urcv x)
       (AU (urcv x)
           (urcvnothing)));

-- Interval when data packet has been received by ABP receiver
-- module and is waiting for receive user to submit a
-- receive request.
-- This interval must have a length of at least one.
defpred data_preserved_interval6 (x)
  (and (eq abp7[1].vars.Recv_buffer_empty^c 0)
       (eq abp7[1].vars.Recv_buffer_datum^c x)
       (urcvnothing)
       (AU (and (eq abp7[1].vars.Recv_buffer_empty^c 0)
                (eq abp7[1].vars.Recv_buffer_datum^c x)
                (urcvnothing))
           (data_preserved_interval7 x)));


-- Interval from when data packet has been delivered to the
-- ABP receiver module  and the receiving user has not
-- processed the previous received data packet yet.
-- This interval may have a length of zero.
defprop exists_y_urcv_y
  (and (eq UAccessPoint_out[1].queued^c 1)
       (eq UAccessPoint_out[1].content[0].kind^c 0));

defpred data_preserved_interval5 (x)
  (and (eq abp7[1].vars.Recv_buffer_empty^c 0)
       (eq abp7[1].vars.Recv_buffer_datum^c x)
       (AU (and (eq abp7[1].vars.Recv_buffer_empty^c 0)
                (eq abp7[1].vars.Recv_buffer_datum^c x)
                (exists_y_urcv_y))
           (data_preserved_interval6 x)));

-- Interval when the data packet is retransmitted while the
-- ABP receiver module's Receive buffer is empty.  This interval
-- must have a length of at least 1.
defpred arcv_x_Send_seq (x)
  (and (eq NAccessPoint[1].queued^c 1)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.id^c 0)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.data^c x)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.seq^c
           abp7[0].vars.Send_seq^c));

defprop exists_y_arcv_y_Send_seq_minus_1
  (and (eq NAccessPoint[1].queued^c 1)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.id^c 0)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.seq^c
           (sub abp7[0].vars.Send_seq^c 1)));

defpred data_preserved_interval4 (x)
  (and (eq abp7[1].vars.Recv_buffer_empty^c 1)
       (or (exists_y_arcv_y_Send_seq_minus_1)
```

```
            (arcv_x_Send_seq x)
            (arcvnothing))
        (AU (and (eq abp7[1].vars.Recv_buffer_empty^c 1)
                 (or (exists_y_arcv_y_Send_seq_minus_1)
                     (arcv_x_Send_seq x)
                     (arcvnothing)))
            (data_preserved_interval5 x)));


-- Interval when the data packet is retransmitted while the
-- ABP receiver module's Receive buffer still contains the
-- previous data packet.  This interval may have a length of zero.
defpred asend_x_Send_seq (x)
  (and (eq NAccessPoint[1].queued^c 1)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.id^c 0)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.data^c x)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.seq^c
           abp7[0].vars.Send_seq^c));

defprop exists_y_asend_y_Send_seq_minus_1
  (and (eq NAccessPoint[1].queued^c 1)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.id^c 0)
       (eq NAccessPoint[1].content[0].DATAinteraction.ndata.seq^c
           (sub abp7[0].vars.Send_seq^c 1)));

defpred data_preserved_interval3 (x)
  (and (eq abp7[0].state^c 0)
       (usendnothing)
       (eq abp7[0].vars.Send_buffer_datum^c x)
       (AU (and (eq abp7[0].state^c 0) (eq abp7[1].vars.Recv_buffer_empty^c
0)
                (or (asendnothing) (asend_x_Send_seq x)
                    (exists_y_asend_y_Send_seq_minus_1)))
           (data_preserved_interval4 x)));

-- Interval when user send request enqueued and waiting for
-- ABP sender module to start processing it
defpred data_preserved_interval2 (x)
  (and (eq abp7[0].state^c 1) (usend x)
       (AU (and (eq abp7[0].state^c 1) (usend x))
           (data_preserved_interval3 x)));

-- Interval in which a previous user send request is
-- being processed and current request is awaiting service.
-- This interval may have a length of 0.
defpred data_preserved_interval1 (x)
  (AU (and (eq abp7[0].state^c 0) (usend x))
      (data_preserved_interval2 x));

defpred data_preserved (x)
  (AG (implies (usend x)
               (data_preserved_interval1 x)));


defprop data_preserved_0 (data_preserved 0);
defprop data_preserved_1 (data_preserved 1);

defprop init_data_preserved_0
  (implies (global_init) (data_preserved_0));

defprop init_data_preserved_1
  (implies (global_init) (data_preserved_1));

defprop forall_x_data_preserved
  (and (data_preserved_0)
       (data_preserved_1));
```

```
defprop init_forall_x_data_preserved
  (implies (global_init)
           (forall_x_data_preserved));

-- Print results
printstring "init_data_preserved_0";
printprop (init_data_preserved_0);
printsize (init_data_preserved_0);

printstring "init_data_preserved_1";
printprop (init_data_preserved_1);
printsize (init_data_preserved_1);

printstring "init_forall_x_data_preserved";
printprop (init_forall_x_data_preserved);
printsize (init_forall_x_data_preserved);
```

Appendix 5: Output of verification

```
Command> Taking input from run_29nov...
init_implies_ag_state_eq_ESTAB_implies_ASendNothing
evaluated (AG 1394ac 0) size=31 totalBDDsize=127650 57:18 mins 4157 Kb
(UAccessPoint_in[0].queued^c
  [1]
  (UAccessPoint_in[0].content[0].kind^c
    [1]
    (UAccessPoint_in[0].content[0].RECEIVErequest.dummy^c
      [1]
      (UAccessPoint_in[0].content[0].SENDrequest.udata^c
        [1]
        (UAccessPoint_in[1].queued^c
          [1]
          (UAccessPoint_in[1].content[0].kind^c
            [1]
            (UAccessPoint_in[1].content[0].RECEIVErequest.dummy^c
              [1]
              (UAccessPoint_in[1].content[0].SENDrequest.udata^c
                [1]
                (UAccessPoint_out[0].queued^c
                  [1]
                  (UAccessPoint_out[1].queued^c
                    [1]
                    (NAccessPoint[0].queued^c
                      [1]
                      (NAccessPoint[1].queued^c
                        [1]
                        (abp6[0].state^c
                          (abp6[0].vars.Send_seq^c
                            [1]
                            (abp6[0].vars.Recv_seq^c
                              [1]
                              (abp6[0].vars.Recv_buffer_empty^c
                                (abp6[0].vars.Send_buffer_empty^c
                                  ...
                                [1]))
                              [1])))
                        [1])))))))))))))))
Unevaluated Size: 28
init_implies_ag_send_buffer_empty_equiv_estab_state
evaluated (AG 3e486c 0) size=35 totalBDDsize=61614 57:24 mins 4157 Kb
[1]
Unevaluated Size: 0
init_ag_not_Send_buffer_empty_implies_Send_buffer_seq_eq_Send_seq
evaluated (AG 3e552c 0) size=29 totalBDDsize=78162 57:29 mins 4157 Kb
[1]
Unevaluated Size: 0
init_ag_state_estab_or_ackwait
evaluated (AG 3e606c 0) size=0 totalBDDsize=78162 57:30 mins 4157 Kb
[1]
Unevaluated Size: 0
init_ag_af_state_ESTAB_and_af_state_ACKWAIT
evaluated (AF 3e666c 0) size=0 totalBDDsize=124310 83:08 mins 4157 Kb
evaluated (AF 3e69ec 0) size=0 totalBDDsize=108789 118:31 mins 4157 Kb
evaluated (AG 3e6aac 0) size=0 totalBDDsize=108789 118:31 mins 4157 Kb
[1]
Unevaluated Size: 0
init_forall_x_one_send_request_at_a_time
evaluated (AU 40962c 409c2c 0) size=32 totalBDDsize=66046 165:45 mins 4189
Kb
evaluated (AG 408dac 0) size=0 totalBDDsize=66046 165:45 mins 4189 Kb
```

```
evaluated (AU 40ae6c 40b46c 0) size=32 totalBDDsize=70902 213:10 mins 4189
Kb
evaluated (AG 40a5ec 0) size=0 totalBDDsize=70902 213:10 mins 4189 Kb
[1]
Unevaluated Size: 0
init_forall_x_repeat_transmit_packet
evaluated (AU 1442ec 1453ec 0) size=79 totalBDDsize=94992 92:53 mins 4189 Kb
evaluated (AU 143f6c 1442ac 0) size=79 totalBDDsize=127221 119:50 mins 4189
Kb
evaluated (AG 1436ec 0) size=0 totalBDDsize=127221 119:50 mins 4189 Kb
evaluated (AU 1462ec 1473ec 0) size=79 totalBDDsize=74929 157:39 mins 4189
Kb
evaluated (AU 145f6c 1462ac 0) size=79 totalBDDsize=77048 184:38 mins 4189
Kb
evaluated (AG 1456ec 0) size=0 totalBDDsize=77048 184:38 mins 4189 Kb
[1]
Unevaluated Size: 0

(( output from test of 06 dec 1993 ))
(( program previous restarted with CPU 103:07 mins. accumulated before this
test ))
Command> Taking input from run_06dec...
init_new_forall_x_s_ack_rcvd_leads_to_incr_seq_num_and_estab
evaluated (AF 40b66c 0) size=0 totalBDDsize=84255 151:15 mins 0 Kb
evaluated (AG 40a72c 0) size=0 totalBDDsize=84255 151:15 mins 0 Kb
evaluated (AF 40cc6c 0) size=0 totalBDDsize=106744 200:20 mins 32 Kb
evaluated (AG 40bd2c 0) size=0 totalBDDsize=106744 200:20 mins 32 Kb
evaluated (AF 40e26c 0) size=0 totalBDDsize=123197 248:54 mins 32 Kb
evaluated (AG 40d32c 0) size=0 totalBDDsize=123197 248:55 mins 32 Kb
evaluated (AF 3e78ac 0) size=0 totalBDDsize=120039 298:18 mins 32 Kb
evaluated (AG 40e96c 0) size=0 totalBDDsize=120039 298:18 mins 32 Kb
[1]
Unevaluated Size: 0
init_new_forall_x_s_send_cycle
evaluated (AU 3f2eac 3f5eec 0) size=66 totalBDDsize=69565 348:06 mins 128 Kb
evaluated (AU 3f172c 3f2e6c 0) size=102 totalBDDsize=124396 383:02 mins 128
Kb
evaluated (AG 3f0cec 0) size=79 totalBDDsize=104255 387:37 mins 128 Kb
evaluated (AU 41076c 4137ac 0) size=66 totalBDDsize=109530 438:36 mins 160
Kb
evaluated (AU 3f6fec 41072c 0) size=102 totalBDDsize=105066 474:20 mins 160
Kb
evaluated (AG 3f65ac 0) size=79 totalBDDsize=115778 479:04 mins 160 Kb
evaluated (AU 41602c 3f906c 0) size=66 totalBDDsize=113697 529:35 mins 160
Kb
evaluated (AU 4148ac 415fec 0) size=102 totalBDDsize=119707 565:08 mins 160
Kb
evaluated (AG 413e6c 0) size=79 totalBDDsize=105884 569:51 mins 160 Kb
evaluated (AU 3fb92c 3fe96c 0) size=66 totalBDDsize=122377 621:27 mins 160
Kb
evaluated (AU 3fa1ac 3fb8ec 0) size=102 totalBDDsize=92572 657:53 mins 160
Kb
evaluated (AG 3f976c 0) size=79 totalBDDsize=129949 662:43 mins 160 Kb
[1]
Unevaluated Size: 0

(( test run on 09 dec 1993 ))
(( program restarted here ))
Command> Taking input from run_09decB...
init_usend_matches_send_buffer_fill
evaluated (AU 40b06c 40b1ec 0) size=59 totalBDDsize=92726 142:42 mins 4253
Kb
evaluated (AU 40b3ac 40b5ac 0) size=40 totalBDDsize=80530 203:52 mins 4253
Kb
evaluated (AU 3fa7ec 40b5ec 0) size=10 totalBDDsize=74233 232:57 mins 4253
Kb
```

```
evaluated (AU 3faa6c 3fac6c 0) size=60 totalBDDsize=96120 292:01 mins 4253
Kb
evaluated (AU 3faeec 3fb0ec 0) size=38 totalBDDsize=104638 353:30 mins 4253
Kb
evaluated (AG 3fb2ac 0) size=10 totalBDDsize=105520 353:34 mins 4253 Kb
[1]
Unevaluated Size: 0
init_send_buffer_fill_matches_seq_number
evaluated (AU 3fc36c 3fc7ec 0) size=35 totalBDDsize=146881 439:27 mins 7101
Kb
evaluated (AU 3fbaac 3fbeac 0) size=35 totalBDDsize=128849 521:22 mins 7101
Kb
evaluated (AU 4094ec 40966c 0) size=36 totalBDDsize=155532 605:27 mins 7101
Kb
evaluated (AU 3fdd6c 3fe1ec 0) size=35 totalBDDsize=148037 689:14 mins 7101
Kb
evaluated (AU 3fd4ac 3fd8ac 0) size=35 totalBDDsize=161202 774:11 mins 7133
Kb
evaluated (AU 3fceec 3fd06c 0) size=36 totalBDDsize=212023 859:36 mins 7133
Kb
evaluated (AG 3fe7ac 0) size=0 totalBDDsize=212023 859:36 mins 7133 Kb
[1]
Unevaluated Size: 0
init_receive_buf_filled_matches_seq_no
evaluated (AU 6d36ac 6c1a6c 0) size=32 totalBDDsize=196910 963:59 mins 7133
Kb
evaluated (AU 6d306c 6d346c 0) size=31 totalBDDsize=224682 1054:22 mins 7133
Kb
evaluated (AU 6d2aac 6d2c2c 0) size=32 totalBDDsize=351280 1155:27 mins
12765 Kb
evaluated (AU 6c2dac 6c316c 0) size=32 totalBDDsize=451911 1252:41 mins
12765 Kb
evaluated (AU 6c276c 6c2b6c 0) size=31 totalBDDsize=430945 1334:37 mins
12765 Kb
evaluated (AU 6c21ac 6c232c 0) size=32 totalBDDsize=502851 1435:52 mins
12765 Kb
evaluated (AG 6c58ac 0) size=0 totalBDDsize=502851 1435:52 mins 12765 Kb
[1]
Unevaluated Size: 0
init_urcv_matches_recv_buff_fill
evaluated (AU 4c2ac 6a502c 0) size=53 totalBDDsize=417658 1545:31 mins 12765
Kb
evaluated (AU 6a51ec 6a53ec 0) size=31 totalBDDsize=316709 1654:01 mins
12829 Kb
evaluated (AU 6a562c 6a542c 0) size=0 totalBDDsize=359758 1682:14 mins 12829
Kb
evaluated (AU 6c60ac 6c62ac 0) size=53 totalBDDsize=505725 1793:24 mins
12829 Kb
evaluated (AU 6c64ac 6c66ac 0) size=30 totalBDDsize=330621 1902:12 mins
12829 Kb
evaluated (AG c3e82c 0) size=0 totalBDDsize=330621 1902:12 mins 12829 Kb
[1]
Unevaluated Size: 0
Command>

(( test run on 12 dec 1993 ))
Command> Taking input from run_12dec...
init_data_preserved_0
evaluated (AU c81eec c824ec 0) size=31 totalBDDsize=474048 1962:36 mins
13021 Kb
evaluated (AU c8112c c8166c 0) size=78 totalBDDsize=450272 2179:49 mins
13117 Kb
evaluated (AU c803ec c80bac 0) size=32 totalBDDsize=486156 2391:16 mins
13181 Kb
evaluated (AU c7e8ac c7ffec 0) size=248 totalBDDsize=332950 2542:02 mins
13181 Kb
```

```
evaluated (AU c7b86c c7d12c 0) size=245 totalBDDsize=919959 2788:32 mins
24349 Kb
evaluated (AU c7ab2c c7b32c 0) size=170 totalBDDsize=714383 2963:48 mins
24349 Kb
evaluated (AU c79aec c7a2ec 0) size=113 totalBDDsize=957107 3111:03 mins
24349 Kb
evaluated (AG c7946c 0) size=119 totalBDDsize=653332 3116:35 mins 24349 Kb
[1]
Unevaluated Size: 0
init_data_preserved_1
evaluated (AU c8b42c c8ba2c 0) size=31 totalBDDsize=998219 3175:29 mins
24349 Kb
evaluated (AU c8a66c c8abac 0) size=78 totalBDDsize=815171 3386:21 mins
24349 Kb
evaluated (AU c8992c c8a0ec 0) size=32 totalBDDsize=865586 3588:13 mins
24477 Kb
evaluated (AU c87dec c8952c 0) size=248 totalBDDsize=672335 3731:32 mins
24477 Kb
evaluated (AU c84dac c8666c 0) size=245 totalBDDsize=772383 3974:43 mins
24477 Kb
evaluated (AU c8406c c8486c 0) size=170 totalBDDsize=716458 4154:13 mins
24477 Kb
evaluated (AU c8302c c8382c 0) size=113 totalBDDsize=736906 4308:55 mins
24477 Kb
evaluated (AG c829ac 0) size=119 totalBDDsize=994813 4314:34 mins 24477 Kb
[1]
Unevaluated Size: 0
init_forall_x_data_preserved
[1]
Unevaluated Size: 0
Command>
```

Appendix 6: Example counterexample trace

```
Command> printtrace (global_init)
(global_nextstate)
                    (and (eq
abp6[0].state^c 1) (not
(asendnothing)));
Start of printtrace.
Clock 319678/100 seconds.
Memory 0 bytes.
Garbage Collecting...
BDD 42152 nodes.

Evaluating start condition...
Done evaluating start condition.
Start condition size = 28 or 28
Clock 319718/100 seconds.
Memory 0 bytes.
Garbage Collecting...
BDD 42152 nodes.

Evaluating goal/invariant...
Done evaluating goal/invariant.
Goal/invariant size = 2 or 2
Clock 319733/100 seconds.
Memory 0 bytes.
Garbage Collecting...
BDD 42152 nodes.

Starting reachability...b0=gcf, as
hypothesized.
Sizes:
    whole = 28
    diff = 28
(32)b0=gcf, as hypothesized.
Sizes:
    whole = 32
    diff = 32
(71)b0=gcf, as hypothesized.
Sizes:
    whole = 71
    diff = 70
(172)b0=gcf, as hypothesized.
Sizes:
    whole = 172
    diff = 156
(319)b0=gcf, as hypothesized.
Sizes:
    whole = 319
    diff = 293
(518)b0=gcf, as hypothesized.
Sizes:
    whole = 518
    diff = 475
(683)b0=gcf, as hypothesized.
Sizes:
    whole = 683
    diff = 635
(825)b0=gcf, as hypothesized.
Sizes:
    whole = 825
    diff = 729
```

```
(1041)A POSSIBLE END STATE
(invariant violated/goal reached):
clock_tick^c = 0
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 1
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 1
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = estab
        .vars
            .Send_seq^c = 1
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 1
            .Recv_buffer_empty^c = 0
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 0
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 1
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 1
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = ack_wait
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 0
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 1
                .counter^c = 0
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 1
            .Recv_buffer_empty^c = 0
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 0
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 1
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 1
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = ack_wait
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 0
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 1
                .counter^c = 3
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 1
            .Recv_buffer_empty^c = 0
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 0
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 1
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = ack_wait
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 0
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 1
                .counter^c = 3
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 1
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 1
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = ack_wait
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 0
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 1
                .counter^c = 2
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 1
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 1
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = ack_wait
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 0
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 1
                .counter^c = 1
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 1
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 1
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = ack_wait
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 0
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 1
                .counter^c = 0
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 0
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 1
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0
```

```
A POSSIBLE PREDECESSOR STATE:
clock_tick^c = 0
UAccessPoint_in
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .RECEIVErequest
                    .dummy^c = 0
                .SENDrequest
                    .udata^c = 0
UAccessPoint_out
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .kind^c = 0
                .SENDconfirm
                    .dummy^c = 0
                .RECEIVEresponse
                    .udata^c = 0
NAccessPoint
    [0]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
    [1]
        .base^c = 0
        .queued^c = 0
        .content
            [0]
                .DATAinteraction
                    .ndata
                        .id^c = 0
                        .conn^c = 0
                        .data^c = 0
                        .seq^c = 0
abp6
    [0]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
    [1]
        .state^c = estab
        .vars
            .Send_seq^c = 0
            .Recv_seq^c = 0
            .Recv_buffer_empty^c = 1
            .Recv_buffer_datum^c = 0
            .Recv_buffer_seq^c = 0
            .Send_buffer_empty^c = 1
            .Send_buffer_datum^c = 0
            .Send_buffer_seq^c = 0
        .timers
            .rexmit_timer
                .running^c = 0
                .counter^c = 0
user_txdatum^c = 0
user_rxdatum^c = 0

End of printtrace.
Clock 322971/100 seconds.
Memory 0 bytes.
BDD 110710 nodes.
Garbage Collecting...
BDD 42152 nodes.
Next state relation uses 7144 bdd
nodes.

Command>
```

Appendix 7: Ever command file for corrected formula

```
-- Ever command file for evaluating new version of formula
-- that states that no send data requests are submitted to
-- the underlying link by the ABP sender module while it
-- is in the ESTAB state.  (Sat 04 Dec 1993)

-- Note, only suitable for model of underlying link with
-- only data packet delivery and data loss.

-- Proposition for exists y,s. ASend(y,s).
defprop exists_y_s_asend (eq NAccessPoint[1].queued^c 1);

-- Define consequent of formula in case need to produce counterexamples.
defprop consequent
    (AU (or (exists_y_s_asend)
            (AU (and (eq abp6[0].state^c 1)
                     (asendnothing))
                (eq abp6[0].state^c 0)))
        (eq abp6[0].state^c 0));

defprop estab_implies_no_asends_initiated
    (AG (implies (eq abp6[0].state^c 1)
                 (consequent)));

defprop init_estab_implies_no_asends_initiated
    (implies (global_init) (estab_implies_no_asends_initiated));

printstring "init_estab_implies_no_asends_initiated";
printprop (init_estab_implies_no_asends_initiated);
printsize (init_estab_implies_no_asends_initiated);

-- Do following in case previous output not a tautology.
printstring "global_init and estab_implies_no_asends_initiated";
printprop (and (global_init) (estab_implies_no_asends_initiated));
```

Appendix 8: Ever output for corrected formula

```
init_estab_implies_no_asends_initiated
evaluated (AU 3e6bac 3e6e6c 0) size=17 totalBDDsize=124058 77:53 mins 0 Kb
evaluated (AU 3e6f2c 3e71ec 0) size=16 totalBDDsize=124469 102:09 mins 0 Kb
evaluated (AG 3e762c 0) size=41 totalBDDsize=85852 103:07 mins 0 Kb
[1]
Unevaluated Size: 0
global_init and estab_implies_no_asends_initiated
!(UAccessPoint_in[0].queued^c
  [1]
  (UAccessPoint_in[0].content[0].kind^c
    [1]
    (UAccessPoint_in[0].content[0].RECEIVErequest.dummy^c
      [1]
      (UAccessPoint_in[0].content[0].SENDrequest.udata^c
        [1]
        (UAccessPoint_in[1].queued^c
          [1]
          (UAccessPoint_in[1].content[0].kind^c
            [1]
            (UAccessPoint_in[1].content[0].RECEIVErequest.dummy^c
              [1]
              (UAccessPoint_in[1].content[0].SENDrequest.udata^c
                [1]
                (UAccessPoint_out[0].queued^c
                  [1]
                  (UAccessPoint_out[1].queued^c
                    [1]
                    (NAccessPoint[0].queued^c
                      [1]
                      (NAccessPoint[1].queued^c
                        [1]
                        (abp6[0].state^c
                          (abp6[0].vars.Send_seq^c
                            [1]
                            (abp6[0].vars.Recv_seq^c
                              [1]
                              (abp6[0].vars.Recv_buffer_empty^c
                                (abp6[0].vars.Send_buffer_empty^c
                                  ...
                                    [1])
                                  [1])))
                          [1])))))))))))))))
```