An Investigation and evaluation of promela/spin as a validation tool for asynchronous concurrent systems

Mark Bezdany

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
AN INVESTIGATION AND EVALUATION OF PROMELA/SPIN AS A VALIDATION TOOL FOR ASYNCHRONOUS CONCURRENT SYSTEMS

by

Mark Bezdany

A thesis submitted in partial fulfillment of the requirements for the

Master of Science Degree in Computer Engineering
Rochester Institute of Technology

Approved by:

Michael J. Lutz, Professor, Department of Computer Science

Tony H. Chang, Professor, Department of Computer Engineering

Roy S. Czernikowski, Professor, Department of Computer Engineering

May 1997
Rochester Institute of Technology

Abstract

AN INVESTIGATION AND EVALUATION OF 
PROMELA/SPIN AS A VALIDATION TOOL FOR 
ASYNCHRONOUS CONCURRENT SYSTEMS

by Mark Bezdany

Chairperson of the Supervisory Committee: Professor M. J. Lutz
Department of Computer Science

Historically, the consequences of implementing faulty designs of concurrent/distributed systems have been well known. There have many documented occasions where the little-known and unaccounted-for situations have caused the loss of human life and limb. This problem can be generalized, nowadays, to systems termed as mission critical. This term has arisen because individuals, businesses, and governments have come to depend on their correct operation. Failures in these systems can have such an adverse impact, that they are simply unacceptable. Although, due to the inherent complexity of these systems, preventing such failures can prove to be a very difficult task.

PROMELA/SPIN is a validation environment that was developed to address the issue of correctness in concurrent systems by means of formal verification. PROMELA is a specification language used to model the systems to be analyzed, while SPIN is a model-checking tool used to perform the analysis. The modeling language, PROMELA, was specifically designed and intended for specifying communications protocols. The tool, SPIN, has the ability to perform both simulations and verifications of a given PROMELA model. It also can perform a bit-state space analysis for maximum coverage of large systems that would otherwise be unable to be exhaustively verified.

This document is the result of an analysis of PROMELA/SPIN as a practical formal verification method. Formal methods have been slow in their development and acceptance because of both the complexity of the problems that they have tried to solve, and the knowledge of formal methods needed to use them practically. With these points in mind, the analysis will explore the types and sizes of systems that can be verified with PROMELA/SPIN effectively, and just how much knowledge of the tool and formal methods are needed to do so.
I, Mark Bezdany, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

Mark Bezdany

21 May 57
Date
# TABLE OF CONTENTS

1 Introduction  ....................................................... 1
   1.0 Some Background on Formal Methods ..................................... 1
   1.1 Where PROMELA/SPIN Fits In ........................................... 4
   1.2 An Overview of This Thesis ............................................ 5

2 PROMELA  .......................................................... 7
   2.0 Introduction .......................................................... 7
   2.1 The Structure of a PROMELA Model ....................................... 8
   2.2 Syntax and Semantics of the Three Basic Elements .......................... 11
   2.3 Syntax and Semantics of Control Flow .................................... 20
   2.4 Correctness Claims ....................................................... 26
   2.5 Comments and Conclusions ............................................. 30

3 Modeling in PROMELA .................................................. 32
   3.0 Introduction .......................................................... 32
   3.1 Modeling Hardware ...................................................... 33
   3.2 Modeling Protocols ...................................................... 37
   3.3 Comments and Conclusions ............................................. 49

4 SPIN’s Functionality .................................................... 50
   4.0 Introduction .......................................................... 50
   4.1 Simulations with SPIN ................................................... 51
   4.2 Verifications with SPIN .................................................. 56
   4.3 The Graphical User Interface XSPIN ..................................... 61
   4.4 Comments and Conclusions ............................................. 63

5 How SPIN Works ........................................................ 65
   5.0 Introduction .......................................................... 65
   5.1 An Overview of the Simulator ............................................ 66
   5.2 An Overview of the Verifier ............................................. 66
   5.3 Verifying Correctness Properties ....................................... 72
   5.4 Comments and Conclusions ............................................. 75
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Block diagram of a simple system</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>K-map for a D-Latch</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic depth-first search algorithm</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Nested depth-first search algorithm</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>Normal operation of the Commit Protocol</td>
<td>78</td>
</tr>
<tr>
<td>6.2</td>
<td>Initial structural representation</td>
<td>81</td>
</tr>
<tr>
<td>6.3</td>
<td>Revised structural representation</td>
<td>84</td>
</tr>
<tr>
<td>6.4</td>
<td>Final structural representation</td>
<td>85</td>
</tr>
<tr>
<td>6.5</td>
<td>Message format—a 7-bit one-dimensional array</td>
<td>87</td>
</tr>
<tr>
<td>6.6</td>
<td>Voting information array—<em>votes</em></td>
<td>88</td>
</tr>
<tr>
<td>6.7</td>
<td>Resource information array—<em>dataTable</em></td>
<td>88</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>PROMELA operators</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Simulation options for SPIN</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Legal LTL Translator operators</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Verification options for SPIN</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Verification runs of the AB Protocol</td>
<td>61</td>
</tr>
</tbody>
</table>
GLOSSARY

Abstraction. A representation that contains only pertinent detail.

Acceptance. A state with an undesirable property that occurs infinitely often.

Assertion. A correctness claim expressed as a Boolean condition that must be satisfied.

Behavioral Specification. A claim encompassing some property that must be checked in the system specification.

Bit-State Analysis. A frugal random validation of a model that cannot be exhaustively verified due to the size of the state space and limited system resources. It attempts to provide maximum coverage while using a minimum amount of resources.

Blocked. Not executable.

Concurrency. The ability to perform multiple tasks in parallel.

Counter-Example. An explicit representation of an error in the system.

CPU. Central processing unit.

Deadlock. When all processes in a system are blocked, and the system is not in a valid end-state.

Executable. A statement that is conditionally true.

Exhaustive Analysis. A full exploration, with or without reduction, of the state space of a model that formally verifies some property.

Formal Specification. The description of a system using a language with a formal semantics.

Formal Verification. A rigorous mathematical proof of the system.

Full-Duplex. Bi-directional information transfer.

GUI. Graphical user interface.
HI. A representation for $V_{cc}$ in discrete logic circuits.

**Instantiation.** The act of running a process.

**Interleaving.** Concurrency modeled on a single processor or control mechanism. This is achieved by allowing control (statement executions) to switch between active processes.

**Logical Clock.** A clock that does not keep "real" time, however, it can be used for the temporal ordering of events.

**Livelock.** Infinite loops in a system that prevent progress.

**Liveness.** A property that shows the absence of livelock.

**LO.** A representation for ground in discrete logic circuits.

**LTL.** Linear-Time Temporal Logic.

**Model Checking.** A verification process that checks properties in a given specification by exploring its state space.

**Non-Deterministic.** Unknown or random.

**On-the-Fly Verification.** A model checking technique that attempts to prove correctness properties while exploring the state space.

**Partial-Order Reduction.** Reducing the number of states to be visited by combining independent statement executions into a single execution sequence.

**Piggy-Backing.** Placing acknowledgments in data packets.

**Point-to-Point.** Direct communication.

**Process ID.** A unique instantiation number for a process.

**Progress-State.** A state with a desirable property that occurs infinitely often.

**PROMELA.** Protocol Meta-Language.

**RAM.** Random Access Memory.
**Rendez-vous Communication.** Synchronized communication between two processes.

**Safety.** A property that shows the absence of deadlock.

**Side-Effect.** An unwanted change in the state of the system.

**SPIN.** Simple PROMELA Interpreter.

**State Space.** Every possible reachable state of a system.

**State Space Caching.** A method of verification that uses disk storage to overcome the state space explosion problem.

**Stutter-Closed.** Executing the last state of a process infinitely often to meet the semantic requirements of LTL for infinite sequences of states.

**System Specification.** An operational description of a system.

**Temporal Claim.** A behavioral specification that specifies some desirable or undesirable behavior of a system.

**Temporal Model Checking.** A model checking technique that attempts to prove correctness properties after exploring the state space.

**Theorem Proving.** Using some temporal logic, a set of axioms, and a set of inference rule to prove properties of a given system.

**Validation.** The process of building confidence in a design or implementation through testing, simulation, or verification.

**Verification.** A formal proof of properties of a system.

**Weak-Fairness.** A priority based mechanism that ensures a non-blocked process will eventually make progress.(execute a statement).

**XSPIN.** A graphical-user interface for SPIN.
INTRODUCTION

1.0 Some Background on Formal Methods

Consequences of implementing faulty designs of hardware and software systems have been well known. There are many documented occasions where the little-known and unaccounted-for situations have caused the loss of human life and limb. Individuals, businesses, and governments have come to depend on the correct operation of these mission critical systems for their life and livelihood. Failures in these systems can have such an adverse impact, that they are simply unacceptable. Compounding the current problem of ensuring correctness is the fact that as time progresses, the size and complexity of these systems will inevitably increase. As one would expect, this also increases the chance that subtle errors—with possibly catastrophic effects—will creep into the final implementation.

Formal methods offer promise in managing this increasing complexity through a number of techniques. Formal methods are mathematical methods used for the specification and verification of systems through the use of languages and tools. Formal specification is the process of modeling a system using a language with a formal (mathematical) syntax and semantics. This can also include the formal specification of known à priori properties of the system. While formal specification, by itself, does not prove a design to be correct, the act of precisely representing the system provides greater insight into its operation.

Formal specifications can represent many different aspects of the system; traditionally, however, they take the form of behavioral representations. For
our purposes here, a specification—be it formal or informal—will consist of two parts, a system specification and a behavioral specification of the system. A system specification is a behavioral or operational description of the system. A behavioral specification is a description of some desirable or undesirable property to which the system specification must adhere.

Formal verification puts to use tools to formally analyze a system for desirable or undesirable behaviors. There are two widely accepted paradigms in the realm of formal verification-- theorem proving and model checking. Theorem proving requires that both the system specification and the behavioral specification be represented by logic formulae. Then, through a set of axioms and inference rules an automated theorem prover will attempt to prove the behavioral specification against the system specification. This method allows for the formal verification of systems with infinite state spaces.

Model checking is the second formal verification method, and can itself be divided into two techniques. The first approach can be termed temporal model checking. It requires a finite state model of the system and a behavioral specification of the system represented in a temporal logic. Once the state space of the system is constructed, in memory or on disk, a check is performed to ensure that the system models the specified behavior. This requires two passes, one to construct the state space and one to the check behavior.

The other approach is termed on-the-fly verification. This requires both the system and the behavior to be represented as finite state automaton. In one pass, usually by a depth-first search of the state space, the behavior is checked against the system. The distinction between the two approaches has been blurred by the development of algorithms which convert temporal logic formulae into finite state automaton.
Both model checkers and theorem provers have their advantages and disadvantages. Model checkers are, by nature, completely automated, whereas theorem provers sometimes are not. A model checker is an algorithmic exploration of some finite state space. Given enough time and resources, it will always produce an answer. Theorem provers reduce the problem of proof to that of a search. It is not as simple as the model checking case, however, in that they are not guaranteed to reach a conclusion. It is entirely possible that the set of axioms or inference rules fall short in their ability to construct a proof, or that the search simply continues with no upper-bound.

The greatest advantage of model checkers is, perhaps, their ability to produce counter-examples. Theorem provers simply can not do this. A counter-example is produced as a direct result of a discovery of an error during a verification run. Because all of the states leading to that error are known and available, it is possible to retrace them to find the origin of the error. This explicit representation of subtle errors can prove invaluable in understanding how they come about so they can be avoided in the future.

The greatest disadvantage of model checkers is the necessity to contend with the state space explosion effect. Even though they only deal with finite state spaces, these spaces can be very large and must be explicitly represented. Both model checking and theorem proving are computationally complex, however, model checkers have the added demand for large amounts of storage. It should be noted that in recent years, advances in CPU performance (computational speed) have far surpassed advances in storage performance (memory size and disk access time). For this reason, other methods for dealing with state space explosion have arisen. Some of these methods are partial-order reduction, localization of information, and state space caching [1].
What then is the intended goal of formal methods? Formal methods are not a cure-all for the problems that exist in systems today. Their goal is simple: to aid in the construction of more reliable systems. Advocates and opponents have made arguments for and against their use as an integral part of the development process. Many of these arguments from both sides, in the face of the facts, would not hold up under formal scrutiny. Formal methods are not intended to replace existing methodologies, but rather to complement them. As an integral part of the development process, they should be able to aid in design and implementation without getting in the way.

1.1 Where PROMELA/SPIN Fits In

PROMELA/SPIN is considered to be an on-the-fly verifier. PROMELA is an acronym for Protocol Meta-Language, and SPIN is an acronym for Simple PROMELA Interpreter. PROMELA is the specification language used to represent both the system specification and the behavioral specification as finite state automaton. PROMELA is not a formal specification language, however. It does not have a well-defined formal semantics or meaning—in fact it has no defined formal semantics. This means that the tool, SPIN, has the final word on the meaning of the language.

SPIN is the validation tool that analyzes PROMELA models for some specified property. I sometimes use the term validation, as opposed to verification, to indicate that SPIN can do much more than just an exhaustive search of the state space. Verification, as defined above, is always a formal proof of correctness. Validation, on the other hand, can be a formal proof of correctness, or it can be an informal method (i.e. testing or simulation) of establishing greater confidence in the system. SPIN can operate as a simulator, an exhaustive state space analyzer, or a bit-state space analyzer. It also has a graphical user interface extension to make it much more user friendly.
The simulator has three modes of operation and is very useful at both uncovering obvious bugs and at analyzing counter-examples. It can perform random simulations that can provide a good view of the operation of the system during the initial design phase. It can perform interactive simulations to explore suspected problem areas of a design. And it can perform guided simulations to uncover the cause of an error, which produced a counter-example during a verification run.

The verifier has two basic modes of operation. The first is an exhaustive state space analysis of the system. If the size of the system being modeled and the resources of the system performing the analysis are conducive to an exhaustive search of the state space, then the tool will rigorously prove (or disprove) the desired properties. If not, then a bit-state space analysis can (probably) be performed. This allows for maximum coverage of a state space that can not be exhaustively verified within the limits of the resources of the system performing the analysis. A bit-state space analysis is not a conclusive proof of the system, however, it still has the ability to find counter-examples.

1.2 An Overview of This Thesis

PROMELA/SPIN was originally designed for protocol verification. Some people believe that verification tools, like SPIN, should be very general in their applicability, while others believe that a tool should only address a very specific subset of verification problems. With this in mind, chapters 2 and 3 explore the modeling language PROMELA. Chapter 2 illustrates the syntax and operational semantics of the language. Chapter 3 then investigates how this language can be applied to the modeling of different types of systems. It also addresses the notion of abstraction in modeling systems with PROMELA—a very important topic when contending with the state space explosion problem.
The next two chapters, chapters 4 and 5, deal with the tool itself. In chapter 4, the functionality of SPIN is explained and illustrated using the models developed in chapter 3. Chapter 4 addresses the modes of the simulator and verifier, and the graphical user interface, XSPIN, from a user's perspective. This entails examples of simulation and verification runs, as well as investigating added functionality in both SPIN and XSPIN. Chapter 5 addresses the theoretical foundations of the tool at an algorithmic level. It does not attempt to establish lemmas or proofs of the underlying formalisms—this is not necessary for the evaluation. It is assumed that the underlying formalisms are sound, and that the language and tool adhere to them. The algorithms, however, that the tool is based on have a direct impact on performance and system resources needed, warranting their investigation.

Chapter 6 documents an example problem from start to finish—a tutorial, if you will. The example problem will be that of the Commit Protocol [2], which is used quite extensively in distributed databases. The final chapter contains comments and conclusions based on the experiences with PROMELA/SPIN in the preparation of this thesis. Further information, documentation, and the source code pertaining to PROMELA/SPIN can be found in Gerard Holzmann's book, [3] in the References of this thesis, or at its home page, http://netlib.bell-labs.com/netlib/spin/whatspin.html.
Chapter 2

PROMELA

2.0 Introduction

PROMELA, an acronym for Protocol Meta-Language, is intended to be a modeling language, not a programming or implementation language. This means that a system specified in PROMELA should be an abstraction of the intended system and only encompass relevant information pertaining to process behavior and interactions. As a result, PROMELA has no floating-point arithmetic, no elaborate abstract data types, and only a few basic types. The compactness and simplicity of the language is indicative of its intended ability to suppress detail. These attributes also lend to the ability of the language to model systems at different levels of abstraction.

The syntax of PROMELA is based on the C programming language. This makes it very easy for most people to understand and use within a very short time. Also, the semantics of PROMELA are based on the guarded command languages of Hoare and Dijkstra. This chapter explores PROMELA in its entirety by, at times, alluding to C or some other more well known programming language.

The most basic constructs in PROMELA programs are tokens, and there are five different classes of them. The first class is identifiers, which are represented by a letter, period, or underscore followed by one or more letters, periods, underscores, or digits. This means that an identifier cannot start with a digit. The second class is keywords, which are reserved words that represent some sort of predefined PROMELA data type, construct, or function. The third class is
constants, which are a sequence of one or more digits that represent an integer. Constants can be defined in one of two ways: via `mtype` definitions (to be discussed later), or with C-style macros. The fourth class contains operators, which are reserved symbols used to perform functions on variables, constants, and channels. The final class is `statement separators`, represented by `;` (the semicolon) or `->` (the arrow). It should be emphasized that they are statement separators and not statement terminators, however, it is not considered an error to use them as either.

The next section discusses the structure of PROMELA models. Section 2.2 explores the syntax and semantics of the language that are directly related to the basic elements—processes, channels, and variables. After that, the syntax and semantics of the different methods of control flow are investigated. This is followed by a discussion of the correctness claims used in PROMELA. The final section is comments and conclusions on the language's ability to model systems, and some of the difficulties and pitfalls that can be encountered.

2.1 The Structure of a PROMELA Model

Models in PROMELA consist of processes, variables, and channels. Processes are always global entities that represent the behavior of the system. Variables and channels can be global or local entities that represent the state of the system. Each of these will be discussed later in detail, however, they are introduced now as a starting point to understand how a model is put together.

---

1 C-style macros and includes are handled by the C-preprocessor and should follow all rules pertaining to the preprocessor being used.
The figure above represents a simple system modeled in PROMELA. All of the processes shown in the figure above are global and can be seen throughout the system. The fact that Process3 and Process4 are inside of Process2 only means that they are instantiated by Process2, but once instantiated they are global entities. The variable, globalVariable, and channels, channel1 and channel2, are all global. Because the two channels are being used by both Process1 and Process2, and assuming that they were not declared within another process and then passed in as parameters, they must be global entities also. The local variables, localVariable1 and localVariable2, were declared within Process1 and Process2 and are therefore local to their respective processes. The only thing left is channel3 and it is assumed to have been declared in Process2, making it local.

Concurrency is modeled, in the PROMELA model above, through interleaved statement executions. The last three words in the last sentence will be an important guide for those who wish to make useful models using PROMELA. This is due to the nature of SPIN as a model checker, which performs verifications by exploring the entire state space. It is necessary to keep the
state space as small as possible so that it can be fully explored. Without keeping this in mind during the modeling process, the end result could be a very good specification of the design that cannot be verified due to the size of the state space. To show how this relates to interleaved statement executions, each of these terms must first be defined and understood separately.

A statement in PROMELA is the most basic executable construct in the language. They are atomic, meaning that if a statement starts execution, it can not be preempted or blocked and must run to its completion. As an aside, there is no difference between the terms, statement and condition, as they are used in this document. Both are treated exactly the same by the verifier. A condition is simply a statement in the form of a proposition. A distinction is made between them here only to make explanations of PROMELA constructs and example models more understandable to the reader.

All statements in PROMELA are executed based on their executability. As explained above, the execution of a statement is the act of successfully passing a statement to its completion. Before this can happen, however, the statement must be evaluated to determine if it can execute—this is how an assignment statement can be the same as a condition. If upon evaluation of the statement, a non-zero value is returned—meaning the condition/statement is not false—then it is executable and will execute. If it is not executable, then it cannot execute and causes the process that contains it to block until it becomes executable.

Interleaving, from a modeling perspective, can be understood partly as a control construct. Because a statement/condition must be executable, it can be used to block a process until it is true—as opposed to polling some condition in a loop using sequential programming methods. When one process blocks waiting for a condition to be true, other processes can continue executing and
may eventually cause that condition to become true. In this way the execution of a process can be controlled through the way in which sequences of statements are put together.

A PROMELA model can now be thought of, structurally, as a system of concurrently executing sequential machines. As promised earlier, this can be related to the size of the state space. This relation is due to the many possible interleavings of statement executions of the sequential machines. Formal verification requires that every possible interleaving be explored.

2.2 Syntax and Semantics of the Three Basic Elements

Systems are modeled in PROMELA as concurrently executing processes that communicate in one of two ways: via message passing or shared variables. In order to understand how this takes place, this section investigates in detail the three basic elements of a PROMELA model—processes, channels, and variables. In the examples below, the following conventions will be followed:

- all keywords will be bold-face type
- process names will begin with an upper-case letter
- variables will begin with a lower-case letter
- constants will be all upper-case letters

Example:

```plaintext
myLabel:          /* This is a comment. */
    printf("This statement is at 'myLabel'";
```

Before moving on to discuss the syntax and semantics of the language, three minor points should be addressed that will aid in understanding the following examples. They are all illustrated in the above example. Comments in PROMELA are basically the same as in C and may not be nested. A label in PROMELA is an identifier followed by a colon, which must be directly
followed by a statement. Output in PROMELA is done using the `printf` statement and is used the same way as in the C programming language.

Processes in PROMELA specify the behavior of the system. They contain a sequence of statements in their declarations and each model must declare at least one process. The way in which a process is declared dictates how it is instantiated: Every process must be instantiated before it can execute. The keywords `proctype`, `init`, `active`, and `run` are used in the declaration and instantiation of processes.

Example:

```promela
proctype DoesNothing( )
{ skip }

active [2] proctype DoNothing(bit x)
{ skip }

init
{ run DoesNothing( ) }
```

All of the three constructs in the above example define processes. The first one uses the keyword `proctype` to define the process `DoesNothing`. There are no parameters to be passed in to this process and it only has one statement that does not do anything. A `skip` statement, like those in the process definitions above, is a pseudo-statement that has no effect on the system but is sometimes needed for syntax requirements. The `proctype` declaration only specifies the behavior of `DoesNothing` processes; however, some other process must instantiate it before it can actually run. This is done here in the `init` process.

PROMELA provides two ways to start processes upon initialization—`init` and `active` processes. If an `init` process declaration is present, then it is always started upon initialization of a simulation or verification run. Consequently,
the **init** process can have no parameters passed in, and there can never be more than one present in any system specification.

The **active** keyword is also used to start up processes upon system initialization. The difference is that the **active proctype** declaration can be used just like any other **proctype** declaration. Because they are instantiated upon system initialization, the issue of parameters must be addressed. In the above example, the parameter, \( x \), is automatically passed in as a 0—as are all parameters for **active** processes. Upon system initialization, two *DoNothing* processes are started—notice the bracketed 2 in the declaration. As all three of these initialization processes are started, they will be assigned process ids 0 to 2 in the order that they happen to be instantiated.

Assume the **init** process is instantiated first, given it a process id of 0. Then the two *DoNothing* processes will be instantiated with process ids 1 and 2. The last process to be instantiated will be the *DoesNothing* process. At this point, there are three instantiated processes in the system with one statement each.

Assume that the *DoNothing* process with the process id of 1 now executes its **skip** statement. It is at the end of its code, but it is not allowed to terminate because there is another process in the system with a higher value process id—so it just waits. Now the other *DoNothing* process executes its **skip** statement. After this occurs, it is allowed to terminate because it has the highest value process id in the system. This causes the first *DoNothing* process to terminate and release its process id. Now there is only one instantiated process in the system, the **init** process. At this time it executes the **run** statement, and creates a new *DoesNothing* process with a process id of 1. In this way, process ids are recycled in a last-in first-out fashion. Once the *DoesNothing* process executes its **skip** statement, it terminates and allows the **init** process to do the same.
As stated before, processes consist of a sequence of statements and a statement consists of one or more identifiers. These identifiers can be *variables* of predefined or user specified data types. There are five basic data types: **bit**, **bool**, **byte**, **short**, and **int**. As one would expect, they look and act the same as their C counterparts—to include having a size that is architecture dependent. Any of these five keywords can be used to declare variables of their respective types just as they are used in C. A key difference between PROMELA and C is that variables are always initialized to 0 by default.

Another data type in PROMELA is introduced with the keyword, **mtype**. This data type is used to specify a set of symbolic names used throughout the program. There can only be one **mtype** definition (set of symbolic names) in each model. Once the set of **mtype** identifiers is defined, they can be declared and used throughout the program like any of the basic data types.

The advantages of using a symbolic name over one of the basic data types are twofold. First, models tend to make much more sense when meaningful names can be used in place of some numerical value—this is especially true when sending messages over channels. Second, because the symbolic names really represent a constant value rather than a variable, the state space of the system can be reduced.

PROMELA also supports arrays and structures. Only one-dimensional arrays are available; two-dimensional arrays can be simulated using arrays of structures, however. Structures are declared using the keyword **typedef**. Aside from the different keyword, the syntax for both arrays and structures is the same as in C.
Example.

```c
mtype = { TYPE1, TYPE2, TYPE3 }; /* 1, 2, 3 */
typedef Structure {
    bit myBit[3] = 1; /* all = 1 */
    bool myBool[3]; /* all = 0 */
    byte myByte;
    short myShort = 0; /* = 0 */
    int myInt = 5;
    mtype myMtype
};
Structure myStruct;
```

The result of the above example is a single instantiated structure, `myStruct`.

The first statement defines the symbolic names in an `mtype` definition for this system specification. The comments on the right show the values given to each symbolic name or variable upon instantiation. Notice that the set of `mtypes` starts at the value 1, and not 0.

The `typedef` definition packages all of the elements declared in it just like structures in C. The only important difference here is the way arrays are initialized. The notation declaring the array `myBit` initializes all of the elements of the array to 1. No loops are needed to initialize the entire array to the same value.

Notice that `myMtype` is initialized to 0, however, there is no corresponding value in the `mtype` definition. If the designer does not deliberately set this field to a meaningful value, then it is not defined as a valid `mtype` in terms of the specification even though it is valid to contain its initialized value of 0.

Variables declared `hidden` are write-only variables. Hidden variables can help reduce the number of global states of the system, because their values are not important and are not considered as part of the system state. Usually, `hidden` variables are not necessary because there is a predefined scratch variable named
by _ (the underscore). The only reason to declare something **hidden** is when you want a meaningful name for a value that you will discard anyway—an unlikely scenario.

Two other predefined variables are used in PROMELA. The first is **_pid**, a local variable to each process containing the process id. No two instantiated processes can have the same value for their respective **_pid**'s. As stated before, however, process ids are recycled upon process termination, so the same value for a **_pid** can be assigned to different processes that are not instantiated concurrently.

The other predefined variable **_last**, is a global variable holding the process id of the last process to execute a statement. This can be very useful in tracking and guiding the interleaving of statement executions.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>greater than</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>less than</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>Modulus</td>
<td>%</td>
<td>equal</td>
</tr>
<tr>
<td>Increment</td>
<td>++</td>
<td>not equal</td>
</tr>
<tr>
<td>Decrement</td>
<td>--</td>
<td>negation</td>
</tr>
</tbody>
</table>

**Table 2.1: PROMELA operators.**

Operations on the above data types are also similar to C. Table 2.1 summarizes the allowable operations.

*Channels* can be either local or global, and can model either synchronous or asynchronous communication. Communication over channels involves the exchange of messages.
Example:
```c
xr myChan1; /* exclusive reads from myChan1 */
xs myChan2; /* exclusive writes to myChan2 */
```

There are two assertions that can be made on channels to facilitate partial-order reduction during system verification. These assertions, placed inside a process, make claim that the process has either exclusive read or exclusive write access to the channel. Such assertions are checked at verification time. This includes checking that only the specified process has the named access, and that operations on the channels are free from side-effects. Side-effects on channels result from using query functions that require both read and write access to a channel.

The messages sent over or received from a channel can be of any type except arrays. The number of fields in a message, and the types that they represent, can have a substantial impact on the size of the global state space. One way to trim down the state space is to be very conservative in the construction of messages to be sent. (i.e., use symbolic names for message fields instead of `bytes, shorts, or ints`)

PROMELA allows incomplete messages to be sent over or received from a channel. When an incomplete message is sent over a channel, the receiver treats an empty field as undefined. When too many message fields are sent over a channel, the receiver simply disregards any extra fields.

Asynchronous communications, or buffered messaging, allows multiple messages to reside in a channel at the same time. The upper-bound on the number of messages that can reside in the channel is specified at the time the channel is defined. The lower-bound for a channel that utilizes asynchronous communication is one. Synchronous communication, or rendez-vous
communication, does not allow any messages to reside in the channel. This type of communication requires a channel to be defined with a zero length.

Example:

```
chan rendezVous = [0] of { myStruct };
chan queue = [10] of { mtype };
```

For the rendezVous channel above, send statements and receive statements must match, requiring process synchronization. Because this operation is synchronous, only two processes can take place in the interaction. If the sender process reaches its send before the receiver reaches its receive, then the sender process will block until the receiver process reaches its receive statement. On the other hand, if the receiver process reaches its receive before the sender reaches its send, then the receiver process will block until the sender process reaches its send statement. It is obvious that there must be some lower-level handshaking going on, however, PROMELA allows for the removal of this unnecessary detail from the model.

Example:

```
myChan!message;    /* a send operation */
myChan!!message;   /* a sorted-send operation */
```

There are two types of sends and two types of receives that can be used in the exchange of messages. A send operation is represented by ! (an exclamation point) and deposits messages into the channel at the tail. A sorted-send operation is represented by !! (two exclamation points) and orders (just the message being sent) in the channel (queue). This is done by starting at the front of the queue and placing the message in front of the first message that has a larger value than itself (a type of ascending order). The value of the message is determined by taking all message fields into account. Also, both types of send operations will block if the channel is full.
Example.
myChan?A_CONSTANT;  /* a conditional receive */
myChan??aVariable;  /* a random-receive */

A receive operation is represented by ? (a question mark) and takes the message at the front of the channel—which it considers to be the oldest message. A random-receive operation is represented by ?? (two question marks) and takes any message currently in the channel randomly. Both of these will block on an empty channel. Receives must store the message value in a variable. Variables of the same type as the message fields or scratch variables can be used. Receive operations can also be made conditional by placing constant values in one or more of the message fields and is illustrated in the example above. By doing this, receive operations will only be executable when the constant values are the same as the values in the corresponding message fields.

Query functions on channels can be used to determine the state of a channel. This information can be used for control flow in the model, or for correctness claims of the model. Some of these functions have side-effects which can cause assertion violations on channels that require exclusive access.

Example.
myChan?[A_CONSTANT];
empty(myChan);       /* true if empty */
full(myChan);        /* true if full */
len(myChan);

All of the operations in the above example have side-effects. Receive operations (?) can be converted from assignments to conditions by enclosing the receiving message in square brackets. This can be very useful in checking the executability of receive operations that are conditional on some constant value. The functions, empty and full, return boolean values pertaining to the
state of myChan. The last function returns the number of messages in the channel at the time the statement is executed. The returned values from any of the above functions can be used for assignment statements or conditions, just as they would in C.

Example:

```c
nempty(myChan);    /* true if not empty */
nfull(myChan);      /* true if not full */
```

The query functions in the above example are the only two functions that are free from side-effects. Therefore, they are the only query functions that can be used in conjunction with exclusive access assertions.

This concludes the description of the three basic components of PROMELA programs—processes, variables, and channels. Every PROMELA model can be understood at a structural level through these three elements. To understand the behavior of a PROMELA model, the semantics of control flow must be understood.

2.3 Syntax and Semantics of Control Flow

Control of statement execution in PROMELA is as important as in any other language. Methods of control can be separated into the following nine distinct types:

- Ordering of statements within a process
- Interleaving of statements among processes
- Jump statements
- Selection statements
- Repetition statements
- Atomic sequences
- Conditional statements
- Escape sequences
- Timeout statements

The first was discussed earlier and pertains to the order in which a sequence of statements is placed in a process. The executability of each statement provides a conditional control mechanism for those statements. If a statement is not executable, then the process will have to block until it becomes executable. Consequently, one statement can be used as a guard for the sequence of statements that follow it.

The second type of control is via interleaving, and is related to the first. A process that is at a statement that is not executable is considered to be blocked. Interleaving (which is how concurrency is modeled in PROMELA) is a way to unblock the process by making the unexecutable statement executable. This is done by the execution of statements in other processes. Another process can alter the state of the system, thus changing the executability of others statements in the system.

Jumps in PROMELA are the easiest of the control flow constructs to understand and are implemented using the goto statement. Jumps are always executable: Whenever they are encountered in a statement sequence, control is transferred to the statement immediately following the destination label. Examples below illustrate jumps with repetition statements and atomic sequences.

Example:

```plaintext
if
  :: (a == b) -> a-- /* if a=b then a=a-1 */
  :: (a == c) -> a++ /* if a=c then a=a+1 */
  :: else -> a = 0 /* if a!=b or c then a=0 */
fi
```

21
Selection statements are similar to if-then-elsif-else constructs found in other languages. The example above illustrates their syntax. The keyword if is used to start the selection statement, and the keyword fi is used to end it. Each :: (double colon) indicates a possible sequence of statements to be selected. The single statement immediately following the double colon is called a guard. In order for a statement sequence to be selected, the corresponding guard must be true (executable). If the first two guards in the above example are not executable, then the else can evaluate to true. An else statement is always blocked when there is another possible choice.

What if both b and c are equal to a? Then there would be two guards that are executable, and one of the two available statement sequences would be chosen at random. It is not important which sequence is taken initially because in the exploration of the entire state space, all possible paths will be explored anyway.

Example.

```
    do
    :: (a == b) -> break
    :: (a == c) -> goto aLabel1
    :: else -> b--; c++
    od;
    a++; goto aLabel2;
aLabel1:
    a--;
aLabel2:
    skip;
```

Repetition statements are syntactically similar to selection statements. Rather than exiting when reaching the od keyword, the statement continually iterates. Upon entering the loop above, the guards are evaluated for executability. If a is equal to b, then the loop will terminate as a result of the break statement, and control will transfer to the a++ statement. After incrementing a, a jump will occur to the skip statement immediately following aLabel2. If a is equal to
\( c \), then a jump will occur to the statement immediately following \texttt{aLabel} and eventually proceed through the \texttt{skip} statement. If \( a \) does not equal \( b \) or \( c \), then the \texttt{else} guard becomes executable. At this time \( b \) is decremented, then \( c \) is incremented, and the repetition statement is reevaluated.

\textit{Atomic sequences} are used to model test-and-set actions and to reduce the global state space. Such statements begin with the keywords \texttt{atomic} or \texttt{d_step}, and are immediately followed by a sequence of statements enclosed in braces. The difference between the two types of atomic sequences, is that the sequence declared using \texttt{d_step} must be deterministic, while the one declared using \texttt{atomic} does not have to be. For a sequence to be deterministic, it cannot have any jumps to outside of the sequence and only the first statement inside the sequence is allowed to block. The distinction between the two is made, because during verification deterministic sequences are executed more efficiently.

\textit{Example.}

\begin{verbatim}
atomic {
if
  :: (a == b) -> goto aLabel
  :: (a != b) -> a++
fi
};
d_step {
if
  :: (a == b) -> skip
  :: else -> a++
fi
};
aLabel:
  skip;
\end{verbatim}

In the above example, if a true guard is evaluated, the statement following it is executed immediately. In no way can the current process containing this code be preempted by the interleaving of statements from another process. There is
one exception to this rule. Because atomic sequences can block, another process can execute if one of the statements in the sequence is not executable. When the statement does become executable, however, control is shifted back to the process with the blocked atomic sequence and the sequence is allowed to run to completion as if it were not interrupted. This creates a certain amount of overhead, and is the reason why d_step sequences are more efficient.

Example.

(booleanCondition -> valuel : value2)

Conditional expressions in PROMELA are similar to those in C. They are used as a short-hand if-then-else, and can only contain expressions or conditions. Because of this, they are always side-effect free. In the example above, when booleanCondition is true, the expression evaluates to value1. If it is false, the expression evaluates to value2. Because booleanCondition must be either true or false, the expression is always executable and will not block.

Escape sequences are declared using the keyword unless between statement sequences. Execution begins with the statement sequence preceding unless. Before any and each statement in the first sequence is evaluated, the first statement of the sequence following unless is evaluated. If at any time this first statement evaluates to true, then control is transferred to the second sequence. This causes the first statement sequence to be abandoned, and the first statement of the second sequence to be executed.

Example.

{ do
  :: (a == b) -> skip
  :: (a != b) -> a++
  od }
unless
{ (a == c) -> a = 0 }
The escape sequence above is entered when the first brace on the first line is reached. The first event to occur is the evaluation of \((a == c)\). If it is true, then the next statement to execute would be \(a = 0\). However, assume that \(a\) is equal to \(b\). This would cause an evaluation/execution (they are the same for a condition that is true) of the first guard in the loop. At this time \((a == c)\) is evaluated again to check if control can be transferred to the second sequence. This way, the execution of the loop can be stopped between any statements, not just between iterations.

The timeout statement is the final control mechanism to discuss. It is a reset mechanism that becomes executable only when no other statements throughout the system are executable. If this statement is encountered in an invalid end-state, it can unknowingly allow deadlock to occur in the model. For this reason, it is very important to use this statement conservatively and with a specific purpose in mind.

The decision to implement timeout statements in PROMELA illustrates the trade-off between modeling power and analytical power in modeling languages. At the expense of modeling power, timing issues were deliberately abstracted out of the language. The gain in analytical power comes from a reduction in the state space by at least an order of magnitude. Timeout statements try to regain some of the modeling power that was lost, while not giving up any of the gain in analytical power.

Example.

```proctype watchdog() { 
do:: timeout -> guard!reset 
   od 
}```
The above example [3] illustrates one of the many uses for timeout statements. The defined process is a watchdog mechanism that attempts to reset the system in the event that the system becomes deadlocked. This is done by sending the message *reset* through the channel *guard*. This assumes that some other process in the system is capable of receiving the message, and will take further action to reset the system.

This section served to present the basic notation and usage of control flow constructs in PROMELA. More advanced usage of these constructs can be seen later in larger examples. Compared to most languages they are relatively few and simple. The next section shows how behavior can be checked in the model of the system.

### 2.4 Correctness Claims

There are three ways in PROMELA to make claims of correctness: assertions, state labels, and temporal claims. The topic of building models in PROMELA will be addressed in the next chapter, but it should be kept in mind that the more complex the correctness claim, the greater the size of the global state space of the system. For this reason, whenever possible, the simplest adequate correctness claim should be used.

**Example.**

```plaintext
assert ( booleanExpression );
```

*Assertions* are by far the simplest correctness claim. They do not increase the state space of the system at all. For a negligible increase in run-time overhead, assertions check that a proposition holds at a given execution point. In the example above, if the expression *booleanExpression* is true (non-zero), then the statement has no effect. If it is false (zero), execution is aborted and that assertion statement is flagged as an error.
Example.

```plaintext
endloop:
  do
    :: myChan1?message -> myChan2!message
  od
```

**End-state labels** mark valid end-states, and are important when checking for system deadlocks. By default, a process reaches a valid end-state when it runs to the end of its code. If this is prevented by control flow in the process, however, then a label using the predefined prefix `end` can be used to signify that a process waiting in that state does not contribute to deadlock. In the example above, the infinite loop is designated as a valid end-state; if the label were not present, the state could be flagged as a deadlock state.

Example.

```plaintext
  do
    :: myChan1?message ->
    progressWhenSent:
      myChan2!message
  od
```

**Progress-state labels** are used to find starvation loops in the design. Starvation loops are cyclic sequences of states that can occur infinitely often. If one of the states in the cyclic sequence is labeled as a progress state, then the sequence of states is not considered a starvation loop. In the example above, there may be a considerable number of interleavings of the two statements contained in the loop with other processes in the system. Using the label ensures that cyclic sequences of states that pass through this label will not be flagged as starvation loops. This can be very important for networking protocols that are designed for unreliable mediums.

**Accept-state labels** are restricted to be used within *temporal claims*. Accept-state labels can be thought of as the opposite of progress-state labels. If an
acceptance-state is found in a cyclic sequence of states, it is considered an error.

Holzmann explains temporal claims as monitors for the system specification that was defined earlier. They are defined using the keyword never. The system specification is an asynchronous product of the processes of the constructed model. In contrast, the system combined with a temporal claim can be thought of many synchronous products of all single traces of the state space. Each trace is a depth-first search of the state space. The temporal claim moves through its local state space by trying to match its propositions against each state in each trace of the asynchronous product. If the temporal claim reaches the end of its code or an acceptance-state is found in a cyclic sequence of states, then an error is reported.

A temporal claim is a behavioral specification in the sense that it monitors behavior. A model of the system is a behavioral specification in the sense that it specifies the behavior. Temporal claims can only monitor undesirable behaviors that are considered errors if present. Any time a desirable behavior is to be verified, it must first be converted to the corresponding undesirable behavior.

Each statement in a temporal claim must take the form of a proposition, making it side-effect free. If at any point a temporal claim alters the state of the system by changing variables or sending messages, it invalidates the verification. Because only a single behavior can be checked at a time, not more than one claim can be present during verification.
Example.

/*
 * Formula As Typed: []p
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]p)
 * (formalizing violations of the original)
 */
never {   /* !([]p) */
T0_init:
  if
    :: (l) -> goto T0_init
    :: (! ((p))) -> goto accept_all
  fi;
accept_all:
  skip
}

The above example never claim is provided to familiarize the reader with its basic construction. It was generated using the LTL translator of SPIN. The temporal claim states that it is an error if p is not true indefinitely, from the initial system state. The (!) statement corresponds to an else statement and is only executable if the other guard, (! ((p))), is not executable. If at any time during system execution p is false, the temporal claim will run to the end of its code.

Example.
proctype example( ) {
  . . .
end:
  do
    :: skip
  od
}

never {
  . . .
accept:
  do
    :: skip
  od
}
When any PROMELA process reaches the end of its code, the process does not simply end. PROMELA is based on stuttering semantics, which is necessary because of the link between temporal claims and linear-time temporal logic (LTL). Because LTL formulae can only be defined over an infinite sequence of states, stuttering semantics requires that the last reachable state of a process occur infinitely often. This is guaranteed in PROMELA with an implicit infinite loop at the end of each process. The above example illustrates the equivalent syntax of these loops for both processes and temporal claims.

This section introduced the usage and meaning of correctness claims. It is very important that they be considered as the model of the system is being built to get the greatest possible benefit with the least possible cost. Assertions require the least amount of resources, followed by end-state labels. Temporal claims, accept-state labels, and progress-state labels are definitely the most costly to use.

2.5 Comments and Conclusions

PROMELA is a relatively compact language, even for a modeling language. It builds on a few basic primitives and constructs, and remains a robust and simple language. Because its syntax is based on C, it is very accessible to most programmers. Learning the language itself is fairly simple. Learning to use the language correctly and wisely is a little more difficult: This topic is discussed in the next chapter.

Generally speaking, most of the problems with the correct usage of the language concern side-effects, channels, atomic sequences, and correctness claims. Another problem with the language is the lack of a written formal semantic definition for the language. Therefore, the verifier has the final say on the semantics of the language. In effect, the only way to become familiar
with the semantics of the language is through the tool. The problem with this is part of the same problem that PROMELA/SPIN tries to solve—guessing how things work, instead of proving how things work.
Chapter 3

MODELING IN PROMELA

3.0 Introduction

With any new technology, there are always people who want to see just how far they can push it. PROMELA/SPIN is no exception, and as a result it has been stretched beyond its original limits. PROMELA has been used to model everything from communications protocols to hardware at the gate level. There are many extensions to PROMELA which enhance its modeling ability in areas like client/server applications and real-time systems. Whether models are specified in PROMELA, PROMELA++, PROMELA R/T, PROMELA C/S, or some other dialect, the fact is that some extensions are useful while others are not.

This chapter presents two models that: (1) provide examples of the language, (2) illustrate some of the different types of systems that can be modeled, and (3) show how the same system can have different models. These modes are important for the effective use of SPIN, and the examples cover the more commonly used constructs of the language.

The next section discusses modeling of hardware circuits in PROMELA, using an asynchronous D-Latch as an example. This is followed by a section with two models of the Alternating-Bit Protocol. The chapter wraps up with comments and conclusions on the ability of PROMELA to model these and other systems.
3.1 Modeling Hardware

This section on hardware modeling is based on the paper by Budi Rahardjo [10] that gave an example verification of a simple asynchronous circuit. The paper specifically addressed the problem of verifying that the asynchronous circuit did not contain static hazards. This is a very important topic in asynchronous circuit design, and a tool that can verify the absence of such hazards would be very useful.

For this thesis, a D-Latch was modeled with and without static hazards. The motivation for modeling a D-Latch is that it is commonly used in circuit design. In an attempt to save space in presenting these models, both the non-hazard and hazard cases were combined into a single model because they are identical except for the DLatch process. The K-map for the D-Latch is shown in Figure 3.1.

![Figure 3.1: K-map for a D-Latch.](image)

The dotted line in the K-map represents the consensus term that may cause a static hazard. The boolean equations corresponding to the two latches are:

- No hazard: \( (d \cdot en) + (~en \cdot q) + (d \cdot q) \)
- With hazard: \( (d \cdot en) + (~en \cdot q) \)

The italicized term represents the consensus term.
The first design problem is how to model the lower-level elements of the latch. This was a difficult task, and with good reason—PROMELA was not designed to model hardware. So following the methodology of Rahardjo’s paper, C-style macros were used to define each gate.

```
#define INVERTER(I, O) 0 != I-I -> O = 1-I
#define AND2(I1, I2, O) 0 != I1&&I2 -> O = I1&&I2
#define NAND2(I1, I2, O) 0 != !(I1&&I2) -> O = !(I1&&I2)
#define OR3(I1, I2, I3, O) 0 != I1||I2||I3 -> O =I1||I2||I3
```

It took a while to understand why Rahardjo chose this representation. For each gate, if the new value is not the same as the old value, then the value of the gate’s output is updated. Otherwise, the guard in the macro definition will block and no update will occur. This is the behavior that the DLatch processes are based upon. The problem here is that only simple logical operations can be used in conjunction with the bit data type. This limits the states that can be modeled to HI (1) and LO (0).

```
bit d, en, q, qold, newInput;
/* No Hazard Here */
proctype DLatch(){
    bit 11, 12, 13, 14;
    do
        :: if
            :: INVERTER(en, 11)
            :: AND2(d, q, 12)
            :: AND2(d, en, 13)
            :: AND2(11, q, 14)
            :: OR3(12, 13, 14, q)
            fi;
        newInput = 0
    od
}
/* Hazard Here */
proctype DLatch(){
    bit 11, 12, 13;
    do
        :: if
            :: INVERTER(en, 11)
            :: NAND2(d, en, 12)
            :: NAND2(11, q, 13)
            :: NAND2(12, 13, q)
            fi;
        newInput = 0
    od
}
```

Now that the gates that make up the latches are complete, the behavior of the latches themselves can be specified. The process definitions for both latches
are above. The process definitions below are self-contained. Therefore, they can be used interchangeably.

The \textit{DLatch} processes above iterate in an infinite loop to model the continuous behavior of real asynchronous hardware. The iterations of the loop are controlled by the macros defined above. If there are changes made to the global variables that model input, then an iteration of the loop will occur. This will inevitably cause \textit{newInput} to be set to zero.

\begin{verbatim}
 proctype Stimulus() {
   do :: timeout -> d_step { newInput = 1; qOld = q;
     if :: d = 1 - d;
     :: en = 1 - en;
     fi
   od
 }
\end{verbatim}

The process definition above represents the stimulus to be applied to the latch being verified. The problem here is that in order to know when another input combination can be applied to the latch, it must be known that the output of the latch is stable. This is done by waiting for system deadlock to occur so that the \textit{timeout} statement will become executable, which requires that the latch process block when stable. Notice also that the updates are made as an atomic step. This is necessary to prevent race conditions on the global variables in the \textit{d_step} sequence.

\begin{verbatim}
 init {
   atomic { run Stimulus(); run DLatch() }
 }
\end{verbatim}

The \textit{init} process above simply instantiates the necessary processes to perform the verification. An atomic sequence is used here to reduce the state space of
the system by reducing the number of possible interleavings. Due to a finite upper bound (which is unknown) on the number of processes that can be instantiated in the system concurrently, a \texttt{d_step} sequence could not be used here because the \texttt{run} statements could potentially block. As a general rule, if the essential behavior of the system is not compromised by the use of one of these statements, then they should be used whenever possible.

\begin{verbatim}
never (
  Top:
    do
      :: ((newInput == 0) && (qOld != q)) \rightarrow break
      :: else \rightarrow skip
    od;
    do
      :: ((newInput == 0) && (qOld != q))
      :: ((newInput == 0) && (qOld == q)) \rightarrow break
      :: (newInput == 1) \rightarrow goto Top
    od
)
\end{verbatim}

The \texttt{never} claim above monitors the input and output of the latch to check for hazards. The claim simply monitors the output for more than one change per input. The first loop looks for the first change in the output only after the latch starts updating due to a change in input. If the output changes again while in the second loop, then the claim is matched and an error is reported. If not, and another input is applied, then the claim resets itself.

There are other ways of modeling hardware to avoid some of the issues raised in this section, but these would increase the state space. For instance, the undefined (unknown) state could be added to the model using symbolic types. The problem here is that representing gates with macro definitions would no longer do: Procedures would have to be modeled using processes. This in turn creates a problem for returning values: Should channels or global variables be used? If one or more channels for each basic element were created, how large a circuit could be modeled?
PROMELA has difficulty modeling hardware because it was not designed for this task: It was created as a protocol modeling language. Hardware modeling requires other constructs in a modeling language than those needed by protocols. For example, processes are poor at modeling procedures because channels are needed to return values, and this requires too many system resources. At the very least, PROMELA would have to efficiently handle procedure or function calls in order to reasonably construct hardware models.

3.2 Modeling Protocols

Communications protocols have continually increased in complexity, in part due to their inherent concurrency and multitude of interactions. This inherent complexity is compounded by increasing performance demands. For all of these reasons, it is extremely difficult to guarantee correct protocol operation.

The PROMELA modeling language was created to formally specify protocols and to help eliminate some of these problems. As is shown in the following examples, many of the basic mechanisms of the language were included specifically to handle protocol behavior. While, it is important to model behavior, a language must also be able to model different levels of abstraction. Both of these issues are addressed in this section.

The protocol used as an example is the Alternating-Bit Protocol (ABP). This is a point-to-point protocol, which means that messages are transferred between two known sites. The protocol is full-duplex: Data can flow in both directions. The ABP is also a synchronous protocol, meaning that one acknowledgment must be received before the next message can be sent. Acknowledgments themselves are always piggy-backed on data packets. Finally, the protocol must work in the presence of an unreliable medium.
The first model was specified at a very low level of abstraction, with the intention of using the simulator to better understand the interactions in the model. With this in mind, the *unreliable medium* was explicitly modeled to support injection of known errors into the system. This in turn showed how the protocol recovered from errors.

Usually, the construction of a model would start from a high level of abstraction and work its way down to the limits of the verification tool. The opposite approach is taken here because this first model was not intended to be verified, only simulated. This is one of the major benefits of the SPIN tool—the simulator and verifier are separate.

The description of the model is interleaved with the code as it was originally specified. Consequently, it should be easier to read and understand the model. The code can be extracted in order and simulated/verified using SPIN.

```plaintext
#define STOP_AT 5
#define MAX_TIME 7
#define ERROR_RATE1 256
#define ERROR_RATE2 256
```

The first five lines define constants that the user can set before a simulation run. *STOP_AT* sets the upper bound on the number of messages transmitted in the system. This was used to gracefully terminate the simulation run. Above, it is set to five, so the first process to reach its fifth message will send a termination message.

*MAX_TIME* sets an upper bound for a logical clock used to simulate a timeout. The error rates specify how often an error should be inserted by the medium in either direction.
init {
    chan StoM = [2] of { bit, bit, byte };
    chan MtoR = [2] of { bit, bit, byte };
    chan RtoM = [2] of { bit, bit, byte };
    chan MtoS = [2] of { bit, bit, byte };
    atomic { run SenderReceiver(MtoS, StoM, 1);
      run Medium(StoM, MtoS, RtoM, MtoR);
      run SenderReceiver(MtoR, RtoM, 0) }
}

The init process manages the initialization of the system, instantiating four channels used to communicate between the three processes init starts atomically. The fields of the messages are a control bit, an acknowledgment bit, and a message sequence number. The size of the channels is set at two, however, due to the synchronous nature of the protocol, there will not be more than one message stored in a channel.

```proctype Medium(chan from1, tol, from2, to2) {
    byte errorByte, tempByte;
    bit tempBit1, tempBit2, looseMsg, badMsg;
    errorByte = 1;
}
```

As stated before, the Medium process acts as an unreliable medium that injects errors at a user specified frequency. In this model, the medium can loose or corrupt messages. The first few lines above simply set up some temporary and logical variables to be used by the process.

Top:
```
  do
  :: skip ->
```

Because there is only one Medium process, which must simulate full-duplex transmission, the process has to alternate between the two incoming channels. Its basic operation is to check each incoming channel for messages, then process that message (if there is one) according to the error control
information. The medium then switches to the other channel and performs the same steps in a loop. The only guard of the main loop was intentionally made a skip to make sure it is always executable.

```plaintext
if
  :: nempty(from1) ->
  from1?tempBit1,tempBit2,tempByte
  :: empty(from1) -> goto Reset1
fi;
```

The selection statement above is used to check if there are any messages currently in the channel. The check-and-remove operation need not be atomic because each process in the system has exclusive read and write access to its input and output channels, respectively. If a message is found in the channel, it is saved in temporary storage, otherwise a jump is made to update the error control mechanism for the next iteration.

```plaintext
if
  :: (looseMsg == 1) -> goto Reset2
  :: (badMsg == 1) -> to2!tempBit1,tempBit2,0
  :: else -> to2!tempBit1,tempBit2,tempByte
fi;
```

The preceding selection statement can only be executed if a message was received in the last selection statement. Based on the settings of the error control information for this channel, it will do one of three things. First, if looseMsg is set, then it will dump the message by just ignoring it. If this happens, a check of the next channel will not occur. It will only update the other channels error control information before re-servicing this channel again. Second, if badMsg is set, then the message is passed on with a zero in the sequence number field to indicate that a resend is needed. Finally, if another guard is executable, the message is sent unaltered.
Reset1:
  looseMsg = 0;
  badMsg = 0;
  if
    :: (errorByte%ERROR_RATE1==0) -> looseMsg=1
    :: (errorByte%ERROR_RATE1==0) -> badMsg = 1
    :: else -> skip
  fi;

The statements above update the error information for the channel. The modulus operator is used to inject errors at a known frequency into the exchange of messages. It is important to notice that one of the two errors must occur at the time specified. This is because the else statement only becomes executable when no other guard is. The else statement can be removed so that errors may occur when specified, but this defeats the purpose of knowing when an error will arise.

if
  :: nempty(from2) ->
    from2?tempBit1,tempBit2,tempByte
  :: empty(from2) -> goto Reset2
fi;

The remainder of the Medium process performs the same tasks on the other input channel. There are a few differences that should be recognized.

if
  :: (tempByte == 255) -> goto StopMedium
  :: else -> skip
fi;

if
  :: (looseMsg == 1) -> goto Reset1
  :: (badMsg == 1) -> tol!tempBit1,tempBit2,0
  :: else -> tol!tempBit1,tempBit2,tempByte
fi;
First, the selection statement above looks for the termination message: when it arrives, the medium allows the process to reach its end of code.

    Reset2:
       looseMsg = 0;
       badMsg = 0;
       if
         :: (errorByte%ERROR_RATE2==0) -> looseMsg=1
         :: (errorByte%ERROR_RATE2==0) -> badMsg = 1
         :: else -> skip
       fi;
       errorByte++
       od;
   StopMedium:
       skip
}

The other point is that the counter that is used for error control is updated. This is the reason why the jump to Reset2 is made in the event of a lost message above.

    proctype SenderReceiver(chan in, out; bit first) {
       byte clock, lastMsgIn, msgIn, msgOut;
       bit controlIn, ackIn, controlOut, ackOut;
       d_step {
          if
             :: (first == 1) -> ackOut = 1; msgOut = 1
             :: else -> controlOut = 1; goto Receive
          fi
       };
    }

The last process in the specification—that actually specifies the behavior of the protocol—is the SenderReceiver process. As the name implies, it performs both sends and receives, making this a full-duplex protocol. The channels passed in are used to communicate with the medium. The bit, first, designates which SenderReceiver process should initiate communication. This is necessary due to the synchronous nature of the protocol—data and acknowledgments must
proceed in lock-step for each process. The first few lines use this information to initialize the process and make sure that they start in the correct states.

Send:
```
atomic{out!controlOut,ackOut,msgOut; clock = 0};
```

The send operation for this protocol is simple, requiring no conditional information to be associated with it. The ability to send a new message is controlled by the receive operation. What the Send must do, however, is reset the logical clock each time it sends a new message. To ensure that the message is sent and the clock is reset at the same moment, an atomic sequence is used.

Receive:
```
   do :: nempty(in) -> in?controlIn,ackIn,msgIn; break
:: empty(in) ->
   if :: (clock >= MAX_TIME) -> goto Send
:: else -> clock++
   fi;
   od;
```

The preceding repetition statement is both a receive and timeout mechanism. After sending a message, the process must wait by polling the incoming channel for the expected acknowledgment. If it does arrive, then the loop will terminate and the process will proceed to analyze the acknowledgment. For each iteration where no acknowledgment arrives, one of two things will happen: Either the logical clock will be incremented (if it is less than MAX_TIME), or a timeout occurs and the message is resent.

There are two reasons for using a logical clock. First, the only alternative is the timeout statement, which relies on system deadlock. The second reason is related to the state space explosion problem. While there are real-time
extensions to PROMELA that provide time, the consequence is a state space that is one or two orders of magnitude larger.

The remaining selection statements analyze the incoming message for consistency and validity. Based on some criterion, the selection statements direct the process to the next appropriate step in the protocol.

\[
\text{if} \quad :: \quad (\text{msgIn}==\text{lastMsgIn}) \quad \rightarrow \quad \text{goto} \quad \text{Receive} \\
:: \quad \text{else} \quad \rightarrow \quad \text{lastMsgIn} = \text{msgIn} \\
\text{fi;}
\]

The above selection statement filters out duplicate packets that were sent as the result of a timeout. It checks the new message sequence number against the old message sequence number to see if they are the same. This would indicate that the new message was already received, and that ignoring it would do no harm.

\[
\text{if} \\
:: \quad ((\text{msgIn} > 0) \&\& (\text{msgIn} != 255)) \quad \rightarrow \quad \text{ackOut} = \text{controlIn} \\
:: \quad (\text{msgIn} == 255) \quad \rightarrow \quad \text{out!}0,0,255; \quad \text{goto} \quad \text{Stop} \\
:: \quad \text{else} \quad \rightarrow \quad \text{goto} \quad \text{Send} \\
\text{fi;}
\]

The selection statement above looks for valid messages. These are messages with sequence numbers from 1 to 254, inclusive. In this case, the acknowledgment sent in the next message is set to the control number just received. A sequence number of zero is reserved to indicate that a retransmission is required. In such cases, no internal updates occur and the last message is resent.
A sequence number of 255 represents a termination message, as specified earlier. This causes a termination message to be sent, and used to gracefully terminate the medium while directing this process to the end of its code.

```plaintext
if
  :: (ackIn == controlOut) ->
    controlOut = 1 - controlOut; msgOut++
  :: else -> goto Send
fi;
```

The above selection statement checks the latest acknowledgment against the control number that was sent in the last message. If they do not match, then an error occurred and the last message must be resent. This is different from the previous selection statement in that this one looks for corrupted acknowledgments. If the acknowledgement and control number match, then the control information and message sequence number for the next packet are updated.

```plaintext
if
  :: (msgOut >= STOP_AT) -> out!0,0,255
  :: else -> goto Send
fi;
Stop: skip
}
```

The last selection statement sends a termination message when the maximum number of messages have been sent. When this happens, it allows the process executing the statement to run to the end of its code.

This specification definitely contains more detail than is necessary to verify the protocol is correct. It was intended to create a better understanding of the protocol. It is not necessary, however, to model the medium, different types of errors, message sequence numbers, or a finite upper bound on the number
of messages to be sent. These will all be removed in the second version of the specification (created with exhaustive verification in mind).

```
#define MAX_TIME 7
mtype = { GOOD_PACKET, BAD_PACKET };
```

The verification model has many fewer statements than the simulation model. The goal here was to keep the state space as small as possible while specifying the essential behavior of the protocol. To do this, the logical clock mechanism was kept and the first line of the program sets its upper bound. The second line shows the symbolic names that were added to represent good and bad message packets. This eliminates the need for a sequence numbering scheme.

```
init {
  chan StoR = [1] of { bit, bit, mtype };
  chan RtoS = [1] of { bit, bit, mtype };
  atomic { run SenderReceiver(RtoS, StoR, 1);
            run SenderReceiver(StoR, RtoS, 0);
  }
}
```

The `init` process is very much the same. There is no medium to instantiate, because the two `SenderReceiver` processes send messages directly to each other. The only significant difference is the message format for the channels. Instead of bytes for sequence numbers, symbolic types are used to represent good or bad messages: There is no reason to distinguish between bad data or bad control information. Finally, the size of the channel was set to one.

Setting the verifier to loose messages when a channel is full, shows the synchronous nature of this protocol. The result of such verifications is that no messages are lost. This confirms that the protocol is synchronized, and that each time a message is place in a channel it is removed before another message is sent.
proctype SenderReceiver(chan in, out; bit first) {
  byte clock, msgIn;
  bit controlIn, ackIn, controlOut, ackOut;
  xr in; xs out;
  if
    :: (first == 1) -> ackOut = 1
    :: else -> controlOut = 1; goto Receive
  fi;
}

The SenderReceiver process was revised to represent all protocol behavior including the unreliability of the medium. The first seven lines in the process declaration initialize the process for communication. As before, necessary variables are declared and initialized to their proper values according to first. The big difference here is the line containing the channel assertions. The xr and xs assertions allow for partial-order reduction during verification runs if they are not violated. This reduces the memory requirements by almost an order of magnitude.

Send:
  atomic {
    if
      :: out!controlOut,ackOut,GOOD_PACKET; clock=0
      :: out!controlOut,ackOut,BAD_PACKET; clock=0
      :: skip
    fi;
  };

The atomic sequence following the Send label above models the unreliable medium. Whenever a message is sent, one of three things can happen. First, a good message can be sent and the logical clock reset, which would indicate that the protocol is operating normally. Second, a bad message can be sent and the logical clock reset, which would indicate that a message was corrupted. Third, nothing is sent which would mean that the message was lost.
Receive:
  do
  :: (clock >= MAX_TIME) -> goto Send
  :: else -> clock++
  od
unless {nempty(in) -> in?controlIn,ackIn,msgIn}

The receive operation is modeled using an escape sequence. It makes the
receive conditional on a non-empty channel without side-effects. This is
important because a violation of the channel assertions would terminate a
verification run. The escape sequence also allows for the loop to be preempted
between each statement of each iteration. Therefore, if a message arrives late,
say just before the \texttt{goto Send} statement, the escape sequence will still be
invoked and the message processed.

\begin{verbatim}
if
  :: (ackIn==controlOut && msgIn==GOOD_PACKET) ->
      controlOut = 1-controlOut; ackOut = controlIn
  :: (ackIn!=controlOut && msgIn==GOOD_PACKET) ->
      ackOut = controlIn
  :: else -> skip
fi;
  goto Send
\end{verbatim}

The above selection statement sets the values of the fields for the next message
to be sent. The first sequence means a good message was received with the
correct acknowledgment. Consequently, the control bit is toggled and the
acknowledgment updated. The second sequence means that a good packet was
received with a negative acknowledgment. This causes no change in the
control bit (a resend of the last data), but the acknowledgment is still updated.
In other words, this process will resend its old data with its old control bit, but
it will still acknowledge the last transmission. The last statement causes a
complete retransmission of the last packet in the event that neither of the
other two guards is executable.
This model is considerably smaller than the first one, because it was intended to perform a different function. PROMELA is very good at representing systems at different levels of abstraction; this is one of the greatest strengths of the language. The verification model above could have been made even smaller by removing the logical clock and using the \texttt{timeout} statement instead, while the simulation model could have been specified with even more detail.

3.3 Comments and Conclusions

This chapter had three goals. The first was to introduce the PROMELA language itself. In the three models shown, most of the common elements were used. Overall, PROMELA is a small and simple language that is easy to learn. As with any other language, however, practice and experience provide greater insight into the different and better ways to represent systems.

The second goal was to show the different types of systems that could be modeled with PROMELA. As with most things in this universe, PROMELA is best suited for its intended application area, and less well suited for other domains. The above examples illustrate this. It is important not only to understand the strengths of a technology to use it properly, but also to understand its shortcomings and limitations.

The last goal was to show how PROMELA can model systems at different levels of abstraction. This is very important in modeling languages used to create specifications that are verified by model checkers. Many times, only a part of the entire system can be modeled and verified due to the limits of system resources. A modeling language that is amenable to slight changes in detail can maximize both resource utilization and confidence in the verified system.
Chapter 4

SPIN'S FUNCTIONALITY

4.0 Introduction

This chapter explores the functionality of the SPIN validation tool (version 2.9.6). There have been many changes to SPIN since its inception, and it continues to evolve. Changes in the tool take the form of bug fixes, extensions to functionality, and alterations in the underlying algorithms. The source code and documentation for the entire PROMELA/SPIN/XSPIN validation environment are freely available from Gerard Holzmann at Bell Labs.

SPIN's main functions are to perform simulations and verifications of a PROMELA model. The simulator can perform random, interactive, or guided simulations. The verifier can perform exhaustive searches of the state space with and without partial-order reduction, or it can perform a bit-state space analysis for maximum coverage of the state space of a system that cannot be exhaustively verified. The tool also includes a translator for converting linear-time temporal logic (LTL) formulae into finite state automaton, and a graphical user interface extension called XSPIN.

The next section describes the simulator and all its functions—the three modes of operation and the LTL translator. The following section focuses on how the verifier can be used to perform different types of verifications. Next is an introduction to XSPIN and the added functionality that it brings. The final section presents comments and conclusions. Examples illustrating the tool's functionality are based on the models from the last chapter.
4.1 Simulations with SPIN

The basic command for using the simulator is:

```
spin [-option] . . . [-option] fileName
```

There can be any number of options used for a simulation run as long as they do not conflict with each other. For each run, the last argument is the name of the PROMELA specification file.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Indicates an interactive simulation run.</td>
</tr>
<tr>
<td>t</td>
<td>Indicates a guided simulation run using a generated simulation trail.</td>
</tr>
<tr>
<td>jN</td>
<td>Will skip the first N steps of a guided simulation run.</td>
</tr>
<tr>
<td>nN</td>
<td>Sets the seed of the random number generator to N.</td>
</tr>
<tr>
<td>m</td>
<td>Loose the messages sent to a full queue.</td>
</tr>
<tr>
<td>l</td>
<td>Print the values of all local variables.</td>
</tr>
<tr>
<td>g</td>
<td>Print the values of all global variables.</td>
</tr>
<tr>
<td>s</td>
<td>Print all send operations.</td>
</tr>
<tr>
<td>r</td>
<td>Print all receive operations.</td>
</tr>
<tr>
<td>p</td>
<td>Print all statements executed.</td>
</tr>
<tr>
<td>v</td>
<td>Print all warning information available.</td>
</tr>
<tr>
<td>d</td>
<td>Print a symbol table containing all initial system information.</td>
</tr>
<tr>
<td>V</td>
<td>Print the current version of the tool being used.</td>
</tr>
<tr>
<td>f</td>
<td>Translate an LTL formula into a never claim.</td>
</tr>
<tr>
<td>a</td>
<td>Generate a verifier called pan.c.</td>
</tr>
</tbody>
</table>

Table 4.1: Simulation options for SPIN.

The simulation options are summarized in Table 4.1 above. The table contains the option identifier and a short description of what it is used for. Similar information can be obtained by invoking the help option via `spin -?`. The remainder of this section uses different combinations of options from the table to provide examples for using the tool.
When SPIN is used as a simulator, its default mode is random. A random simulation with no options gives very little information by default. For this reason, options should be used to help check system behaviors. The following example uses the simulation model of the Alternating-Bit Protocol from the last chapter.

Example.

C:\>spin -n9 -p FullDuplexAB_S.pml
   0: proc - (:root:) creates proc 0 (:init:)
   1: proc 0 (:init:) creates proc 1 (SR)
   1: proc 0 (:init:) line 211 "FullDuplexAB_S.pml"
      (state 4) [(run SR(MtoS,StoM,1))]
   2: proc 0 (:init:) creates proc 2 (Medium)
   2: proc 0 (:init:) line 213 "FullDuplexAB_S.pml"
      (state 2) [(run Medium(StoM,MtoS,RtoM,MtoR))]
   149: proc 3 (SR) line 94 "FullDuplexAB_S.pml"
      (state 54) [((ackIn==controlOut))]
   150: proc 3 (SR) line 96 "FullDuplexAB_S.pml"
      (state 50) [controlOut = (1-controlOut)]
   151: proc 1 (SR) line 70 "FullDuplexAB_S.pml"
      (state 29) [((clock>=7))]
   152: proc 2 (Medium) line 136 "FullDuplexAB_S.pml"
      (state 63) [((1)]
   153: proc 2 (Medium) line 138 "FullDuplexAB_S.pml"
      (state 10) [else]
   154: proc 3 (SR) line 97 "FullDuplexAB_S.pml"
      (state 51) [msgOut = (msgOut+1)]

In this example, the statements were printed as a result of the p option. In total, there are 1076 statement executions for the run. All of this information is useless one knows what to look for. The n option was used to alter the seed value for the random simulation run.

Interactive simulation lets the user guide the execution of statements. During an interactive simulation, the system will execute statements until a non-deterministic choice must be made. At that time, the user will be prompted to input the number of the statement to execute next. The following example shows how this works.
Example.

C:\>spin -i FullDuplexAB_S.pml
Select a statement
  choice 1: proc 3 (SR) line 35 "FullDuplexAB_S.pml"
  (state 1) [clock = 0]
  choice 2: proc 2 (Medium) line 130
  "FullDuplexAB_S.pml" (state 1) [looseMsg = 0]
  choice 3: proc 1 (SR) line 35 "fullduplex1.pml"
  (state 1) [clock = 0]
  choice 4: proc 0 (:init:) line 216
  "FullDuplexAB_S.pml" (state 5) unexecutable, [-end-]
Select [1-4]: 2
Select a statement
  choice 1: proc 3 (SR) line 35 "FullDuplexAB_S.pml"
  (state 1) [clock = 0]
  choice 2: proc 2 (Medium) line 136
  "FullDuplexAB_S.pml" (state 63) [(1)]
  choice 3: proc 1 (SR) line 35 "FullDuplexAB_S.pml"
  (state 1) [clock = 0]
  choice 4: proc 0 (:init:) line 216
  "FullDuplexAB_S.pml" (state 5) unexecutable, [-end-]
Select [1-4]:

Interactive simulation runs can be very time consuming, and should only be used when looking for something very specific. The time needed by an interactive run can be reduced by using XSPIN.

The last type of simulation is a guided simulation. These simulation runs are based on a state trail written to disk (in file pan_in.trail) as a result of an error during a verification run. Guided simulations are one of the tool's best features: A claim or property violation can be detected in a verification run, and can be exposed by performing a guided simulation run.

Example.

C:\>spin -t -v -j8 pan_in
  8: proc 1 (SenderReceiver) line 16 "pan_in"
     (state 9) [clock = 0]
  9: proc 1 (SenderReceiver) line 19 "pan_in"
     (state 14) [. (goto)]
spin: line 19 "pan_in", Error: assertion violated
10: proc 1 (SenderReceiver) line 19 "pan_in"
     (state 15) [assert(0)]
spin: trail ends after 10 steps

53
The example above was constructed using the verification model of the Alternating-Bit Protocol. An assert(0) statement was placed in the model to guarantee an error. During verification, an error is produced, the verification stops, and the verifier automatically writes the state trail to file.

Example.

```c
C:\>spin -f "p -> <>q"
never ( /* p -> <>q */
T0_init:
   if
     :: (1) -> goto T0_S2
     :: (\! ((p)) || (q)) -> goto accept_all
   fi;
T0_S2:
   if
     :: (1) -> goto T0_S2
     :: ((q)) -> goto accept_all
   fi;
accept_all:
   skip
)

C:\>spin -f "!(p -> <>q)"
never ( /* !(p -> <>q) */
accept_init:
T0_init:
   if
     :: (\! ((q)) && (p)) -> goto accept_S3
   fi;
accept_S3:
T0_S3:
   if
     :: (\! ((q))) -> goto accept_S3
   fi;
accept_all:
   skip
)
```

The last simulator component is the LTL Translator. As stated in Chapter 2, a never claim models an undesirable behavior. Therefore, when translating an LTL formula into a never claim, the LTL formula must specify a behavior that represents a violation if it is matched. It should be noted, however, that it is
possible to specify desirable behaviors with a **never** claim by negating the LTL formula before translation. The preceding example shows two **never** claims produced by the translator—the second is the negation of the first.

The first claim asserts that it is an error if the following behavior is matched: If \( p \) becomes true at some point during system execution, then \( q \) must eventually become true. The second claim asserts exactly the opposite of this.

Two syntactic rules were followed in devising the LTL formulae. First, predicates must be lower-case, which means that all variable names used in the formula must start with a lower-case letter. Second, there can be no spaces between the characters that make up a single logical operator. Table 4.2 summarizes the legal operators in LTL formulae.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Always</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Eventually</td>
</tr>
<tr>
<td>U</td>
<td>Strong until</td>
</tr>
<tr>
<td>V</td>
<td>Dual of until</td>
</tr>
<tr>
<td>&amp;&amp; or /\</td>
<td>And</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Negation</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Implication</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>Equivalence</td>
</tr>
</tbody>
</table>

Table 4.2: Legal LTL Translator operators.

Simulations are a valuable way to debug PROMELA specifications. They provide feedback to the user (especially through XSPIN) relatively quickly. Random simulations are great at finding the obvious bugs in a newly constructed model. Interactive simulations allow a user to quickly check a suspicious statement sequence. Guided simulations are perfect for pinpointing hard-to-find bugs discovered during a verification run.
4.2 Verifications with SPIN

To perform a verification run, the simulator builds a C program from a PROMELA specification via the `a` option. The C program is then compiled using a number of directives that affect the verification run; a complete list of the compilation directives is in Appendix E.

Many of the compilation directives must be used in conjunction with specific verifier run-time options. Table 4.3 briefly summarizes the verification options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td>Search for acceptance cycles.</td>
</tr>
<tr>
<td><code>b</code></td>
<td>Use backward single-bit hashing.</td>
</tr>
<tr>
<td><code>cN</code></td>
<td>Halts a verification run upon reaching the $N^{th}$ error.</td>
</tr>
<tr>
<td><code>d</code></td>
<td>Prints all of the information in the state tables.</td>
</tr>
<tr>
<td><code>e</code></td>
<td>Saves state trails for all encountered errors.</td>
</tr>
<tr>
<td><code>f</code></td>
<td>Enforces weak-fairness.</td>
</tr>
<tr>
<td><code>hN</code></td>
<td>Allows a user to choose a list of hash functions numbering 1 to 32.</td>
</tr>
<tr>
<td><code>i</code></td>
<td>Search for the shortest error trail.</td>
</tr>
<tr>
<td><code>l</code></td>
<td>Faster and approximate search for the shortest error trail.</td>
</tr>
<tr>
<td><code>mN</code></td>
<td>Search for non-progress cycles.</td>
</tr>
<tr>
<td><code>mN</code></td>
<td>Sets the maximum search depth to $N$.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>Disables the listing of unreachable states.</td>
</tr>
<tr>
<td><code>q</code></td>
<td>Enforces the requirement of empty channels in valid end-states.</td>
</tr>
<tr>
<td><code>s</code></td>
<td>Uses forward single-bit hashing.</td>
</tr>
<tr>
<td><code>V</code></td>
<td>Prints the current version of the tool being used.</td>
</tr>
<tr>
<td><code>wN</code></td>
<td>Sets the size of the hash table to $2^N$.</td>
</tr>
</tbody>
</table>

Table 4.3: Verification options for SPIN.

The default verification run is an exhaustive search of the state space with partial-order reduction. In the example below, the verification model of the Alternating-Bit Protocol illustrates how to search for deadlock.
Example.

$gcc -o pan -DMEMCNT=25 -DSAFETY -DNOCLAIM -DVAR_RANGES
   -DNOFAIR  pan.c
$pan -m6000 -w19 -cl

Full statespace search for:
  never-claim - (not selected)
  assertion violations +
  cycle checks - (disabled by -DSAFETY)
  invalid endstates +

State-vector 48 byte, depth reached 5019, errors: 0
  29341 states, stored
  29001 states, matched
  58342 transitions (= stored+matched)
  22486 atomic steps
  hash conflicts: 17565 (resolved)
  (max size 2\(^{19}\) states)

Stats on memory usage (in Megabytes):
  1.643 equivalent memory usage for states
     (stored*(State-vector + overhead))
     compressed State-vector=37 byte + 8 byte overhead
  1.311 actual memory usage for states (compression: 79.78\%)
  2.097 +memory used for hash-table (-w19)
  0.168 +memory used for DFS stack (-m6000)
  0.221 +memory used for other data structures
  3.699 =total actual memory usage

unreached in proctype SenderReceiver
   line 37, state 37, "-end-"
   (1 of 37 states)
unreached in proctype :init:
   (0 of 4 states)

Values assigned within interval [0..255]:
  SenderReceiver:msgIn : 0-2,
  SenderReceiver:clock : 0-7,
  SenderReceiver:ackOut : 0-1,
  SenderReceiver:controlOut : 0-1,
  Sender Receiver:ackIn : 0-1,
  SenderReceiver:controlIn : 0-1,
  SenderReceiver:first : 0-1,

The directive, \texttt{MEMCNT=25}, sets the upper-bound of allowable memory that the verifier can use at \(2^{25}\), or 32 megabytes of RAM. It is very important—especially on systems with virtual memory—to confine this upper-limit to the available real memory. Otherwise, paging could cause the length of the verification run to become unacceptable. The \texttt{SAFETY} directive indicates a
search for invalid end-states during system execution. NOCLAIM, simply tells
the verifier to ignore temporal claims. VAR_RANGES, computes the values
of all variables within the range of 0 to 255. Finally, NOFAIR disables the
code for weak-fairness, which shortens the verification run.

The first section of output provides feedback on the properties examined by
the verifier. The next two sections show the size of the system both in terms
of states and transitions, and actual memory needed to perform the
verification. The last two sections provide extra information not pertaining to
safety properties.

By default, unreachable code is reported after every verification run. This helps
expose over-specified systems and possible missing functionality. The variable
ranges shown were explicitly requested; these can help spot problems when the
bounds of variables are known during system design.

Memory is often the limiting factor in model checking. In this example the
state vector is 48 bytes, and comprises all of the channels and variables that
contribute to the global state of the system. The verifier automatically
compressed this value to 37 bytes, reducing the total memory required to store
states by 20%. If the automatic compression scheme does not utilize memory
efficiently enough, then the COLLAPSE compilation directive should be used.
Using this directive reduces the memory requirements by 60%, compared with
no compression at all. The cost for the memory savings, however, is two to
three times more run-time.

Most memory is used in the verification run for the hash table that tracks the
visited states. The hash table size is set by the \textit{n} run-time option, which
estimates the state space size. In exhaustive verification runs its importance is
diminished because all collisions in the hash table are resolved by using linked-
lists; this is not the case when performing a bit-state space analysis. The memory required for the stack and data structures is very small compared to the hash table, but the size of the stack should be kept to a minimum to allow for a larger hash table and more states. A series of trial verification runs must be performed to achieve this.

The following example shows the memory requirements after explicitly disabling partial-order reduction.

Example:
State-vector 48 byte, depth reached 58329, errors: 0
576345 states, stored
736707 states, matched
1.31305e+06 transitions (= stored+matched)
442552 atomic steps
hash conflicts: 1.16174e+06 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
29.970 equivalent memory usage for states
(= stored*(State-vector + overhead))
23.12 actual memory usage for states (compression: 77.17%)
2.097 +memory used for hash-table (-w19)
1.440 +memory used for DFS stack (-m60000)
0.422 +memory used for other data structures
26.987 = total actual memory usage

Notice that the state space is over 22 times larger than it was when the partial-order reduction algorithm was applied, which in turn increased memory requirements more than 7 times.

One more use of verification will be explained: bit-state space analysis. Again, the same Alternating-Bit Protocol is used for the example. This is not the best specification for this type of analysis, because bit-state analysis is usually applied to systems that cannot be verified exhaustively. However, the example does show memory requirements and coverage of the bit-state space approach.
Example:
State-vector 48 byte, depth reached 4979, errors: 0
26931 states, stored
26843 states, matched
53774 transitions (= stored+matched)
20681 atomic steps
hash factor: 155.737
(expected coverage: >= 99.9% on avg.)
(max size 2^22 states)

Stats on memory usage (in Megabytes):
1.400  equivalent memory usage for states
       (stored*(State-vector + overhead))
0.524  memory used for hash-array (-w22)
0.160  +memory used for DFS stack (-m5000)
0.221  +memory used for other data structures
1.413  =total actual memory usage

For a bit-state space analysis, it is very important for the hash factor to be
greater than 100 to ensure the best coverage of the state space. The hash
factor is the ratio of the size of the hash table to the number of states stored.
The COVEST compilation directive gives an estimate of the coverage in
percentages—but it is only an estimate. Here the memory savings are not
substantial, however, the savings can be enormous when verifying larger
systems.

It is not always possible to perform a bit-state space analysis. Most of the
literature about bit-state analysis indicates that, at the expense of coverage, one
can always be performed. However, if there is not enough memory to store
the data structures, search stack, and hash table, then the bit-state space
approach will not work.

The table below summarizes the six verification runs for the Alternating-Bit
Protocol. The complete PROMELA specification for this model can be found
in Appendix I.
<table>
<thead>
<tr>
<th>Type of Analysis</th>
<th>Run-Time Option</th>
<th>Used Partial-Order Reduction</th>
<th>Actual Memory Used in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive</td>
<td>N</td>
<td>Y</td>
<td>3.699</td>
</tr>
<tr>
<td>Exhaustive</td>
<td>Y</td>
<td>Y</td>
<td>3.084</td>
</tr>
<tr>
<td>Exhaustive</td>
<td>N</td>
<td>N</td>
<td>26.987</td>
</tr>
<tr>
<td>Exhaustive</td>
<td>Y</td>
<td>N</td>
<td>11.969</td>
</tr>
<tr>
<td>Supertrace</td>
<td>N/A</td>
<td>Y</td>
<td>1.413</td>
</tr>
<tr>
<td>Supertrace</td>
<td>N/A</td>
<td>N</td>
<td>9.896</td>
</tr>
</tbody>
</table>

Table 4.4: Verification runs of the AB Protocol.

For exhaustive verification, partial-order reduction has a tremendous impact on memory requirements. The difference is almost an order of magnitude for the system used in this example. The reason is the way channels are used in the model. It is very important to try to place exclusive access restrictions on channels when possible to gain the full benefit of partial-order reduction.

The bit-state space analysis with partial-order reduction uses the least amount of memory. For a small system like this one, it is very easy to get the hash factor over 100—producing a coverage estimate of over 99.9%. With larger systems and limited resources, it becomes harder to achieve such coverage. For this situation, different techniques can be used to increase confidence in the model. For example, the SPIN verifier is equipped with both forward and backward single-bit hash functions, as well as the ability to use thirty-two different hash functions. This still does not guarantee total coverage, but sixty-four verification runs with independent hash functions is as close as any tool can get without performing an exhaustive search.

4.3 The Graphical User Interface XSPIN

XSPIN greatly simplifies the task of learning the tool: It allows the user to utilize all of the functionality of SPIN, while extending its output capabilities.
The extra capabilities include graphical representations of the processes in the system as state machines, and graphical representations of message exchanges in the form of message sequence charts. It also provides different forms of text output that help to understand the system.

Upon starting XSPIN, the main control window appears. The window has two basic functions: (1) as an editor for the current PROMELA specification, and (2) as an interface to functionality not provided by SPIN. The File button performs the same basic operations that would be found in any application with a GUI. The Help button provides limited advice for the other available functions. The rest of the buttons—Simulate, Verify, LTL, and FSM View—create windows that will be explained below.

The Simulate button produces a window that prompts for simulation options. This includes what to display during the run, the type of simulation to run, and how to handle full channels. There are only two ways to handle sending messages to a full channel: block messages or loose messages. The simulation type can be any of the three mentioned previously—interactive, guided, or random.

There are four display options. The Execution Bar Panel produces a window with a bar graph showing (in real-time) the percentage of statement executions belonging to each process. The Variable Values Panel traces the values of all of the variables in the specification. The Time Sequence Panel shows the interleaved sequence of statement executions in text form. Finally, the Message Sequence Chart shows the sequential ordering of events taking place on system channels. This is a valuable tool when trying to understand communication between processes.
The *Verify* button creates a window that prompts the user for verification options. For verification runs, XSPIN and SPIN differ only in look, not content.

The *LTL* button produces a window that is an interface to the LTL Translator. The interface provides additional features: It can automatically negate a formula when it is a desired property instead of an undesired one. It can also automatically import the generated *never* claim directly into the current specification in the main control window.

Finally, pressing the *FSM View* button creates a window with all process names contained in the specification. Double clicking on a name creates a window with the graphical representation of the process’ finite state machine. This is useful in understanding the construction and operation of individual processes.

4.4 Comments and Conclusions

The simulator provides a great deal of feedback about the system in a relatively short period of time. Its three modes of operation—guided, interactive, and random—give the user a robust and adaptable simulation environment. The graphical information produced by XSPIN makes the large amount of simulation output easier to work with. This in turn reduces the time required to perform simulations.

The verifier takes time to learn, and here XSPIN is not much help. There are three basic issues when using the verifier: (1) the user must determine the correct options to use, (2) the user must ensure that these options do not conflict with one another, and (3) the user must understand the verifier enough to use the best options for a particular run. Needless to say, this requires experience with the tool.
Even an experienced user finds it somewhat difficult to perform verifications of larger systems in the face of limited resources. The tool addresses this by offering different verification options, but using these require many extensive trials. The fact that specifications consuming many system resources usually take much longer to complete further limits the usefulness of these options.
Chapter 5

HOW SPIN WORKS

5.0 Introduction

SPIN was identified in Chapter 1 as an on-the-fly verifier; this chapter tells what that means. As noted previously, SPIN is both a simulator and a verifier. Although the simulator and verifier are distinct, they both perform their tasks on-the-fly.

The next section presents the simulator as a PROMELA interpreter. Because this thesis is more concerned with SPIN as a validation tool, only basic simulator functions are investigated.

The simulator discussion is followed by an overview of the verifier, which explores the three different search methods in the tool: a fully exhaustive search, an exhaustive search with partial-order reduction, and a bit-state space analysis. A single section suffices because all three approaches use the same basic algorithm. A bit-state space analysis, for example, is an exhaustive search that records minimal amounts of information about visited states.

How the verifier handles correctness claims is presented in Section 5.3. Assertions and end-state labels do not require an extended version depth-first search algorithm presented in Section 5.2, however, the inclusion of progress-state labels, accept-state labels, and temporal claims require a nested depth-first search algorithm.

The last section presents comments and conclusions.
5.1 An Overview of the Simulator

This section introduces the basic structure of the simulator. The simulator interprets both complete and incomplete specifications, an invaluable service when constructing new designs and debugging old ones.

SPIN is an acronym for Simple PROMELA Interpreter. SPIN is used as a simulator primarily to provide a better understanding of the specification, and to find the causes of errors discovered by the verifier. Evaluating expressions is at the core of the simulator, and uses the notion of “executability” discussed in Chapter 2.1.

The simulator has a lexical analyzer, a parser, and a scheduler. A PROMELA specification is the input for the lexical analyzer, which breaks the program into lexical tokens. The tokens are passed to the parser, which constructs a parse-tree. Finally, the scheduler performs the simulation by traversing the parse-tree. The simulation proceeds top down by evaluating each node (expression) in the parse-tree.

For a complete description of the algorithms and functionality, consult the latest source code or [3] in the References.

5.2 An Overview of the Verifier

A model checker formally verifies a specification via a reachability analysis of its state space. SPIN’s reachability analysis can take three forms: (1) a completely exhaustive analysis of the state space, (2) an exhaustive analysis of a subset of the state space that retains correctness properties, or (3) a bit-state space analysis called Supertrace. This section describes how the tool performs each search.
All three techniques start with the same basic view of the system. They view each PROMELA specification as a labeled transition system with a finite number of states and transitions. Every transition has both a precondition and an effect. The precondition is based on the local state of the process in which the transition (statement) resides. The effect is the local or global state change made (if any) as a result of the transition (statement execution).

Example:

```plaintext
1  active proctype example( ) {
2    chan sr = [1] of { bit };
3    do
4      :: sr!0
5      :: sr!1
6      :: sr?0
7      :: sr?l
8    od
9  }
```

Unreachable states are unvisited states in the labeled transition system, and fall into two classes. Unreachable local states refer to unused code in the specification, which is reported as dead code by the verifier. This dead code is the result of missing functionality or over-specification. If line 5 were removed from the above example, line 7 would be reported as unreachable code because the statement would never be executable.

Unreachable global states should include all error states, representing the undesirable properties specified by the designer. Generally, the number of unreachable states of a system is much greater than the number of reachable states. This is fortunate for model checking, as algorithms are already constrained by the large number of reachable states. Removing line 6 or 7 from the example above would lead to an undesirable deadlock state, which is currently an unreachable global state.
The fully exhaustive search is simplest. It provides the greatest coverage of the state space, but can only be applied to very small problems due to the number of states that it must explore. Exhaustive analysis is done by a depth-first search of the state space.

procedure dfs(state)
    if state is an error
        then report error and exit
    add state to state_table
    add state to state_stack
    for each successor child_state of state
        if child_state is not in state_table
            then dfs(child_state)
            delete state from state_stack
    end

Figure 5.1: Basic depth-first search algorithm [8].

The algorithm starts by calling the depth-first search procedure with the initial system state. The state passed in is checked for errors; if it is not an error state, then it is added to a hash table that tracks all visited states. It is also pushed onto the search stack that contains all of the states visited in the current pass of the depth-first search. Finally, the depth-first search procedure is called recursively for each successor to the current state.

There are two data structures used in the above algorithm, a stack and a table. The stack, in this algorithm, is only used to show counter-examples. When a state has no successors (the bottom is reached), states are popped from the stack and the algorithm backtracks to a previously visited state with an unvisited successor state.

The table records visited states, and it is organized in memory as a hash table of linked lists. Linked lists house all visited states, and they are used within the hash table to resolve any hash collisions.
The full search method performs more work than is necessary by exploring sequences of states that do not need to be explored. Consequently, it places inflated demands on storage requirements by storing impertinent information. Partial-order reduction methods avoid exploring redundant interleavings by reducing them to a single statement sequence.

Redundant interleavings are statement sequences proven capable of being represented by a single statement sequence. Therefore, if multiple interleavings can be interchanged without affecting correctness properties, it is sufficient to explore only one of them. Partial-order reduction reduces the total number of states to be explored, and is analogous to the use of atomic sequences to reduce the number of possible interleavings discussed in Chapter 3. The difference is that this concept is now applied across processes, instead of being restricted to the statements of a single process.

Example.
```c
byte globalVar;

active proctype Process1( ) {
   byte localVar1;
   localVar1++; /* 1 */
   globalVar = globalVar + 1 /* 2 */
}

active proctype Process2( ) {
   byte localVar2;
   localVar2++; /* 3 */
   globalVar = globalVar - 1 /* 4 */
}
```

In the example above, there are two statements in each process, requiring six different execution sequences to fully explore the state space. The set of statement interleavings is \{ (1, 2, 3, 4), (1, 3, 2, 4), (1, 3, 4, 2), (3, 4, 1, 2), (3, 4, 1, 2), (3, 1, 4, 2), (3, 1, 2, 4) \}. Because statements 1 and 3 access local variables, only two interleavings must be explored that switch statements 2 and
4. The statement sequences \((1, 2, 3, 4)\) and \((3, 4, 1, 2)\), for example, would be sufficient.

Partial-order reduction of the state space can be performed dynamically during the search, or statically before the search. Dynamic reduction strategies require additional memory and CPU time, and because this occurs during the verification run, when system resources are already in high demand, the benefits of performing the reduction may be limited. For this reason, SPIN uses a static reduction strategy.

The static reduction method that SPIN uses follows the notion of statement independence mentioned earlier. Two statements are *independent* if and only if the following conditions are met [5]:

1. They are both enabled.
   
   \(\text{AND}\)

2. The execution of either cannot disable the other.
   
   \(\text{AND}\)

3. The effect of execution is indistinguishable of the order.

Two statements are *globally independent* if and only if the following condition is met:

1. They reside in different processes and access only local objects.

The criteria above for statement independence are not sufficient in the presence of a temporal claim if the effects of the statements are observable to the claim. Consequently, the reduction criteria can make the temporal claim incapable of being violated when it should, allowing an error to go undetected. Solving this requires that only one statement in the sequence is observable to the temporal claim or capable of changing state.
With the above conditions in mind, five statements can be statically marked as either conditionally or unconditionally safe:

1. exclusive send statements
2. exclusive receive statements
3. nfull statements
4. nempty statements
5. any strictly local statement (accesses only local objects)

An unconditionally safe statement cannot be a guard in a control flow mechanism, whereas a conditionally safe statement must be a guard in a control flow mechanism. Consequently, a conditionally safe statement, based on its executability, chooses the statement sequence that follows it; an unconditionally safe statement does not.

Before verification, the processes are reordered to optimize the statically marked statements. Executing conditionally safe statements first minimizes the state space because they dynamically choose statement sequences to follow. Unconditionally safe statements and unsafe statements are executed last. For a complete description of the partial-order reduction algorithm used by SPIN, see [5] and [8].

The final method of validation that SPIN performs is a bit-state space analysis (a.k.a. Supertrace). Supertrace can be thought of as an exhaustive search that does not resolve any hash conflicts: The linked-lists are dropped. Consequently, it cannot guarantee correctness, and is not intended for systems that can be verified by the methods mentioned previously.

Because Supertrace uses the same basic algorithm for state space exploration as above, partial-order reduction also applies here. The effects of partial-order reduction are twofold: The state space that is explored is reduced, and this in
turn increases the coverage of the analysis. The coverage of Supertrace is estimated as the hash factor (see Chapter 4.2).

Each table entry contains a single bit of information: A table entry of 0 indicates that the state has not been visited, while a table entry of 1 indicates that the state may have been visited. Collisions occur when a state is hashed into the table where the bit is already 1. Consequently, the current state must be treated as if it were previously visited, introducing an unknown amount of uncertainty.

The uncertainty of Supertrace is unknown due to the many consequences of hash collisions. A hash collision, for example, will make an unvisited state be treated as one visited. This in turn may cause an unvisited successor of the unvisited state to go unexplored; this depends on the connectivity of the state graph, and is why SPIN can only estimate coverage.

5.3 Verifying Correctness Properties

Three types of correctness claims that can be made in PROMELA take the form of assertions, state-labels, and temporal claims. Assertions are boolean conditions; either true or false. Because they are evaluated like any other statement, they do not require any alterations to the algorithms.

Similarly, end-state labels are easily checked by the verifier, and do not require changes to the depth-first search algorithm to establish their correctness. End-states are checked for validity when all reachable states have been explored. An end-state is valid if each process in the system has reached the end of its code or a labeled end-state.

Checking accept-state labels employs a nested depth-first algorithm to find acceptance cycles. Acceptance cycles are cyclic sequences of states that contain
an accept-state label. Figure 5.2 presents the algorithm used to search for acceptance cycles.

```plaintext
procedure dfs(state)
    if state is an error
        then report error and exit
    add {state, 0} to state_table
    add state to state_stack
    for each successor child_state of state
        if {child_state, 0} is not in state_table
            then dfs(child_state)
        if state is an accepting state
            then ndfs(state)
        delete state from state_stack
    end

procedure ndfs(state)
    add {state, 1} to state_table
    for each successor child_state of state
        if {child_state, 1} is not in state_table
            then ndfs(child_state)
        else if child_state is in state_stack
            then report cycle
    end
```

Figure 5.2: Nested depth-first search algorithm [8].

The `dfs` procedure searches for error states until it reaches a state with no successors, then it backtracks looking for a state with an accept-state label. If an accept-state label is found in the current pass of the depth-first search, the `ndfs` procedure is called with the labeled accept-state and the search for acceptance cycles begins.

The `ndfs` procedure looks for any sequence of states that will lead it back to the accept-state label it started with by re-searching the successor states of the labeled accept-state. The `state_stack` is searched for each `child_state` encountered in the `ndfs`. If a `child_state` is found in the `state_stack`, an acceptance cycle is reported.
The nested depth-first search algorithm performs well, minimally increasing memory requirements because the second search procedure does not have to maintain its own stack. The ndfs procedure simply uses the same search stack as the dfs procedure. Because of redundant searching, the time to complete the search can double if no acceptance cycles are found, but this increase in runtime will usually not preclude verifications from being performed.

Checking progress-state labels is transformed into a search for acceptance cycles. A progress state is a state that immediately follows a progress-state label. Non-progress requires that cyclic statement sequences be reported as errors if they do not contain a progress state. Cyclic statement sequences without progress states are considered starvation loops (livelock), because they may occur infinitely often.

Progress states are considered boolean conditions, which can alter the global boolean variable progress. progress is true if and only if the current state is labeled with a progress-state label. This means that a search for non-progress cycles becomes a search for acceptance cycles by forming a temporal claim of the boolean condition progress.

Example:
```plaintext
never { /* <>[]=][!progress */
T0_init: if
     :: (1) -> goto T0_init
     :: (! ((progress))) -> goto accept_S2
fi;
accept_S2: if
     :: (! ((progress))) -> goto T0_S2
fi;
T0_S2: if
     :: (! ((progress))) -> goto accept_all
fi;
accept_all: skip
}
```
The temporal claim is $\Diamond \Box \neg \text{progress}$, and is automatically generated by SPIN when a search for non-progress cycles is performed. The LTL formula $\Diamond \Box \neg \text{progress}$ means that progress cannot remain false indefinitely. The temporal claim in the above example illustrates the implicit claim generated by SPIN.

Temporal claims are easy to implement, although they have the potential to greatly increase memory requirements and run-time. The current condition (statement) of the temporal claim is checked after each transition of the system to a new state. A match in conditions of a temporal claim results in a test for acceptance cycles due to stuttering semantics; this is discussed in Chapter 2.4. If no acceptance cycle is encountered, the claim cannot be matched. The possible increase in memory due to a temporal claim is the system state space multiplied by the number of matched states of the temporal claim.

5.4 Comments and Conclusions

Using the simulator is not necessary to perform verifications, however, it is extremely helpful when trying to understand the behavior of a model. Added functionality, like simulators, greatly improves the chances of formal methods being accepted and used in the future. Without the simulator, SPIN would fall short as a formal verification tool.

The on-the-fly method of model checking verifies correctness properties while the state space of the model is explored in a single pass. This eliminates the need to construct and store the entire state space so that a check of correctness properties can be performed in a second pass. Adding partial-order reduction and compression to on-the-fly model checking, which SPIN does, ensures that the applicability of SPIN is bound to grow.
Chapter 6

THE COMMIT PROTOCOL

6.0 Introduction

Transaction processing in distributed databases is a difficult problem from both the data consistency and performance perspectives. Data consistency requires that a transaction be processed without interference from any other transactions in the distributed system. Performance in transaction processing systems is measured by the time required to successfully complete a transaction.

Both of these concerns have been addressed in many different ways. The most profound differences in approach lie in the locking mechanisms, which provide exclusive access to data during updates. One method of locking data places most of the burden on the clients. Each client, upon forming a transaction, must locate and lock the resources it requires before any changes are made, thus ensuring exclusive access until the transaction is complete. After all of the locks are acquired, changes are made and the resources released. Without some preemption mechanism, there is no upper bound on the length of time a client is allowed to hold resources, which could degrade performance considerably. Implementing this preemption mechanism will undoubtedly increase network traffic due to the fact that clients must now communicate with each other while competing for resources.

Another method places most of the burden of performing transactions on the servers. In this scheme, a client just sends each transaction to a server to be processed and awaits an acknowledgment. The request contains all of the
information necessary to complete the transaction, and is usually based on replicated data with some versioning or timestamp mechanism to keep the data consistent. After receiving the transaction, the server locks all of the necessary resources to perform the updates. If all of the locks can be put in place and all versions are consistent, then the updates are made and the transaction is completed. If not, then the transaction is aborted and the client may retry by making another request. This allows the server to perform transactions atomically, guaranteeing consistency.

The designs above represent trade-offs between ease of maintaining consistency, and ease of maintaining performance. In the first method, performance depends on the preemption mechanism. The overhead of this preemption mechanism, and any associated priority scheme, inevitably causes a loss in performance. In the second method, data consistency depends on the replication of data and versioning. This method is based on an assumption that in a very large database, the contention for resources will be minimal. When the assumption holds, this method is very efficient. When the assumption does not hold, a considerable amount of overhead is incurred by updating copies of data in the system and retrying transactions.

The Commit Protocol uses the second method described above. This chapter is a tutorial on the formal verification of the Commit Protocol using PROMELA/SPIN. Because it is a tutorial, it attempts to encompass as much pertinent material as possible in a start-to-finish fashion. The verification process, however, is a trial-and-error process much of the time. Consequently, only the more important problems encountered during the verification of this protocol are addressed.

The next section presents an informal specification of the protocol. Sections 6.2 and 6.3 explain the process of building a formal model that is capable of
verification within the available resources. Section 6.4 discusses the PROMELA implementation of the final version of the Commit Protocol specification. The final section presents comments and conclusions.

6.1 The Informal Specification

The Commit Protocol [2] is classified as an all-or-nothing protocol because it either completes or aborts an entire transaction. For this reason, it must operate in two phases. The first phase makes sure that the necessary resources are available and then locks them. If the first phase is successful, then the second phase performs the updates and releases the resources. If the first phase is not successful, then the second phase aborts by notifying the client and the involved servers that the transaction must be aborted. Regardless of whether or not a transaction is completed or aborted, it must go through two phases.

![Diagram of the Commit Protocol](image)

Figure 6.1: Normal operation of the Commit Protocol.

---

1 All verifications of the Commit Protocol in this chapter were performed on HP Model 715 workstations running at 100MHz, and containing 128 MB of RAM.

2 See Appendix J for the complete PROMELA specification of the Commit Protocol.
A sequence of events for the Commit Protocol, in the absence of any faults, is shown in Figure 6.1. It displays the normal operation of the coordinator and worker(s) involved in a transaction. Although the figure only shows one worker, there can be any number of workers involved throughout the network with any single coordinator. The first phase of the protocol includes the first two arrows in the figure, and the second includes the last two.

Although a client may need the resources of more than one server in the network, it is not concerned with locating them. A client only has to choose a server that holds at least one of the resources in the current transaction, and send that server the transaction request. The server that receives the request from the client becomes the coordinator for that transaction.

Upon receiving a new request, the coordinator starts the first phase of the protocol. The first action is to ensure that any resources that it must contribute to the current transaction are available. If one or more resources required by the coordinator cannot be acquired, then the transaction cannot be completed. The coordinator then notifies the client that the transaction was aborted without involving any other servers.

If all of the resources at the coordinator can be acquired, then they are locked for the remainder of the transaction. The coordinator then asks the other servers (workers) holding needed resources if those resources are available by sending each of them a CanCommit message. If a worker can commit the resources, it locks them and returns a Can message to the coordinator. If a worker cannot commit all of the resources needed in the transaction, then it returns a Cant message to the coordinator. When all votes are in from all workers, the coordinator starts the second phase of the protocol.
The events of the second phase are determined by the votes received by the workers in the first phase. If no *Cant* messages were received, the coordinator notifies all workers to make the necessary updates by sending each of them a *DoCommit* message. The coordinator, at this time, also performs an update on the resources that it has locked for the current transaction. If a *Cant* message was received from any worker, then the coordinator tells all workers to abort the current transaction by sending them an *Abort* message. In addition, the coordinator releases any resources that it is holding for the current transaction without updating them. After the workers receive either a *DoCommit* or an *Abort* message, they release all resources involved in that transaction. If the workers receive *Abort* messages, no acknowledgement is sent; however, they receive a *DoCommit* message, they send the coordinator a *HaveCommitted* message. Upon receiving the acknowledgement, the coordinator notifies the client of the transaction's status.

It should be noted that error-free communication is assumed here. If this were not the case, a number of extensions would have to be made to the operation of the protocol shown in Figure 6.1. Consequently, the coordinator would have to provide timeout mechanisms for any expected replies, and a decision-making ability for either retrying or aborting a transaction. Similarly, this scheme does not provide protection from server failures. Even this simplified version, however, is more difficult to verify than it might appear.

### 6.2 Building the Model

The last section described the Commit Protocol's operational behavior. However, before building a model that can be submitted for verification, a structural representation must be decided upon. It is this structure that is used as a guide to form process definitions and the methods of communication between them. Each process is a partial behavioral representation of the
system that can be used to group non-conflicting operations that take place sequentially. Keeping in mind that the size of the state space is the limiting factor during verification, and the protocol specifies that the coordinator also acts as a worker, a very conservative structural representation is possible.

Because a coordinator must perform tasks as both a coordinator and a worker, there is no need to specify them separately. Therefore, it should be sufficient to define just two processes. The \textit{Client} process generates transactions for the coordinators. The \textit{Coordinator} process represents servers in the network.

![Diagram](image_url)

\textit{Figure 6.2: Initial structural representation.}

With only two process definitions, the protocol should be easily verifiable with a modest amount of system resources. This representation is sufficient to create contention for resources and to show the protocol is sound. In the system above, there are two coordinators, two workers, and one client. Each server houses two resources. Communications channels are set up between the client and both coordinators for transaction requests and acknowledgements. Communications channels are also set up between the coordinators and workers. \textit{Figure 6.2} illustrates this structure.

This structural representation is very conservative, requiring only three instantiated processes and six channels. For a number of reasons, however, it
cannot be verified. Also, the structure is over-simplified. Consequently, the model does not fully satisfy the informal specification given in the first section.

The first problem encountered during the verification process concerned the use of atomic sequences. An atomic sequence was placed in the model that contained a loop. The loop contained a non-deterministic choice of assignment statements, and was thought to eventually force a jump to a location outside of the atomic sequence. Even though the loop represented a finite number of states, the atomic statement made it seem like an infinite loop to the verifier.

*Example.*

InfiniteLoop:

```plaintext
atomic {
    . . .
    goto InfiniteLoop
}
```

FiniteLoop:

```plaintext
atomic { . . . }

goto FiniteLoop;
```

The example above illustrates the infinite loop problem. Although the `goto` statements appear to jump outside of the atomic sequence, they are actually jumping to the first statement inside the atomic sequences. This caused the depth-first search stack to grow to capacity and terminate the search. This problem was fixed by placing the `goto` statement that closed the loop outside of the atomic sequence.

A second and unrelated problem concerned the use of compilation directives. Because it seemed that the state space was too large for the available memory, the `REACH` directive was used. This directive should guarantee, in a search truncated by memory constraints, the validity of the search up to the point of termination. When used with exhaustive searches, however, the verification
run never ends. For example, an exhaustive verification run was performed using this directive, and ran for 40 hours without exhausting memory or completing.

The most difficult problem encountered was the inability to verify the model due to memory constraints. Because the model is very small and simple, this was unexpected. It is conjectured that the instantiation of two Coordinator processes was to blame. The theory is that a process definition that takes channels as parameters cannot have multiple instantiations without invalidating the partial-order reduction on those channels. In correspondence with Holzmann, he discounted this theory, but had no other explanation to offer.

System resource problems aside, the model does not satisfy the requirements for the protocol. The use of only two workers in the system is insufficient for modeling the voting phase of the protocol when one of the workers is embedded in the coordinator. The voting phase is over-simplified in the model to the point that it becomes trivial.

If the coordinator can acquire its resources, it sends a CanCommit message to the other worker in the system. In the event that the worker cannot commit, it aborts the transaction and alerts the coordinator. At this point, all servers involved in the transaction know that the transaction was aborted. Therefore, there is never a need for the coordinator to send an Abort message to any workers. The statement that sends the Abort message is reported as unreachable code during verification.

This requires that the model be extended to include at least one more worker. The inclusion of this extra worker forces the coordinator to store information pertaining to the voting process for each transaction. Because the model, in its
current state, is unverifiable, it seems that the added complexity of storing information will make this problem much worse--this is not the case.

6.3 Refining the Model

The design from the last section does not have to be completely abandoned. By making changes incrementally, the effects of the changes can be studied and understood. Even though each change is made individually, a new structural representation should be constructed and used as a goal in each step of the revision process.

![Figure 6.3: Revised structural representation.](image)

The changes made to the model in Figure 6.3 are minor, however, they have an enormous impact on the verification process. Only one change was made to the client: It now communicates with just one coordinator process. The process that acts as both a coordinator and a worker did not have to be changed at all. A new Worker process definition was created using the Coordinator process definition as a template. For the new process definition, however, all functions pertaining to the coordinator were removed. This makes it strictly a worker process.

There are now three process definitions in the model, and only one process is instantiated for each definition during verification. The specification of the
revised model can now be verified without exhausting system resources. In fact, the model went from being unverifiable using over 200 MB of memory to being verifiable using about 25 MB of memory (both attempts used the compression compilation option).

There are two explanations for the reduction in the state space. The first explanation, discussed previously, asserts that multiple instantiations of a process with channel parameters invalidates the partial-order reduction on those channels. If this is true, it may be just a bug in the verifier.

The other explanation is that the elimination of one coordinator in the system reduced the complexity of the model. Although the reduction in the state space was dramatic, this is possible. Applying the input space to two coordinators causes an interleaving of two 25 MB state spaces. Determining conclusively which explanation is correct would require enough memory to run tests that could pinpoint the problem.

![Diagram](image_url)

**Figure 6.4:** Final structural representation.

Figure 6.4 illustrates the changes made to the structure of the model to meet the requirements of the protocol. The addition of another worker in the system makes it necessary to store extra transaction information in the
coordinator, because no assumption can be made about the time each worker will take to process a transaction. This gives the coordinator the ability to send an *Abort* message to one worker when it receives an *Abort* message from the other worker.

Four processes are instantiated and six channels are used for communication during verification of the model. This configuration was verified for *safety* using 85 MB of memory. A verification for safety proves the absence of deadlock in the model by checking for any invalid end-states. It was also verified for *liveness* using just under 100 MB of memory. A verification for liveness proves the absence of livelock in the model by checking non-progress cycles.

The original specification that was unverifiable consisted of about 175 lines of PROMELA code. The final version of the Commit Protocol specification consists of over 400 lines of PROMELA code. This shows that the complexity of the model is not always proportional to the size of its PROMELA program.

6.4 The Final Version

This section discusses the PROMELA specification, corresponding to Figure 6.4, at the code level. Appendix J contains the complete specification of the Two-Phase Commit Protocol.

The first 23 lines of the program define constants, data types, and communication channels to be used throughout the model. The *mtype* declaration on lines 13 and 14 contains nine message types. The six types of messages exchanged between the coordinator and workers are: *CanCommit, Can, Cant, Abort, DoCommit*, and *HaveCommitted*. The three types of messages exchanged between the client and coordinator are: *NewTrans, Committed*, and *Aborted*. These message types are used for conditional receives throughout the model and represent the first field of a message.
In five out of the six channels, the second message field is represented by the structure, message. It contains an array of seven bits. The first six bits identify the requested resources for the transaction, while the last bit is the transaction identification number. This is illustrated above in Figure 6.5. For the first six bits, a 1 indicates a request for that resource. The transaction ID can have the value 0 or 1, consequently there can be no more than two transactions in the system concurrently.

The Client process (lines 24-68) continually loops through an escape sequence. This is done to give preference to messages coming from the coordinator. If there are no messages from the coordinator and there are less than two transactions in the system, the client attempts to send a transaction request. The client will not always be successful in doing this because in order for a transaction to be sent, it must request at least one resource from each server. This constrains the input space to the more meaningful transactions while reducing the state space of the model.

An atomic sequence is used to protect the local information in the Client process pertaining to outstanding transactions. This is done to prevent the escape sequence from changing control before line 55 can be reached and the new transaction sent. Otherwise, deadlock would eventually occur because the client would think that the transactions were sent and would await status information that would never arrive.

The Coordinator process (lines 69-270) is the largest and most complex process definition in the specification. Most of this can be attributed to the dual
function the coordinator performs; both as a coordinator and as a worker. Some of this complexity is also results from the need to communicate with the client and workers.

The main loop (lines 77-120) of this process also uses an escape sequence. The escape sequence is used here to give preference to transactions already in the system by handling messages from workers first. However, if a new transaction is accepted from the client, the coordinator’s processing of the new transaction cannot be preempted by a message from a worker. Preferences aside, the coordinator can take any message type at any time, by using transaction identification numbers and a local memory to coordinate events.

![Diagram of Voting Information Array](image1)

Figure 6.6: Voting information array--votes.

The coordinator stores information to keep track of votes and resources. An array of eight bits is used to house voting information for the transactions in the system. The format for the information in the array is shown in the Figure 6.6 above. If any of the questions in the figure can be answered with a yes, a 1 is placed in that location in the array. Otherwise, the value in that location remains a 0. The information can be accessed when needed to coordinate events within the coordinator.

![Diagram of Resource Information Arrays](image2)

Figure 6.7: Resource information arrays--dataTable.

Figure 6.7 illustrates the information stored to lock resources. Two bits of storage information would be sufficient for each resource, however, three bits
were used for two reasons. First, control flow is easier to implement. Second, this makes it much easier to compose a temporal claim that can monitor exclusive access to the resources.

There are two transaction processing functions in the Coordinator process definition—COHandle1 and COHandle2. COHandle1 (lines 121-158) processes new transaction requests from the client. It first checks for the availability of the requested resources in the server that encompasses the coordinator. If all of the resources are available, locks are placed on them and the workers are sent CanCommit messages. If one or more of the requested resources are unavailable, the client is notified that the transaction was aborted.

Before the locks are implemented, assert statements ensure that the lock was not set by another transaction (see lines 136 and 144). This eliminates the need to use a temporal claim to monitor exclusive access to resources which reduces the memory requirements to verify the model.

COHandle2 (lines 159-269) processes all messages sent by workers. It uses the stored voting information discussed earlier and two bits that indicate the current phase of a transaction to process messages. If the transaction is in the first phase and all voting information is acquired, then either DoCommit or Abort messages are sent to the appropriate workers in the system as required by the protocol. The client will also be notified at this time with an Aborted message if the transaction was aborted. If the transaction is in the second phase of the protocol, then once all commits have been acknowledged, the client is notified that the transaction has been committed.

The worker processes, W0 and W1 (lines 273-350 and 351-427, respectively), are functionally equivalent. They are very simple because they need to process only three different types of messages from the coordinator. If a CanCommit
message is received, the worker uses the same method of locking and resource information storage as the Coordinator process definition. The only difference here is that the worker immediately responds to the coordinator with a message that indicates its ability to carry out the transaction.

If the worker receives a DoCommit or Abort message from the coordinator, the actions taken are almost exactly the same. Because this model is an abstraction of the protocol, no real data is used and no real updates take place. Therefore, the reception of either of these two messages simply indicates that the worker should release all locks on resources pertaining to the current transaction. The difference between the handling of the two messages above is that a HaveCommited message must be sent to the coordinator if the transaction was committed.

The next section examines the verification process of the Commit protocol specification discussed above. This focuses on two parameters: memory requirements and run-time.

6.5 Comments and Conclusions

The specification in Appendix J was verified for both safety and liveness properties. During the revision process, it was necessary at times to allow the verifications to use part of virtual memory. This caused a significant increase in run-time, and is not recommended. However, it sometimes is the only way to verify the model.

The following example illustrates the commands, compilation directives, and partial results of a verification run for safety properties for the Commit Protocol specification. Explanations for all options and directives can be found in the Appendix. The important directive to notice here is COLLAPSE. It invokes a compression algorithm that reduces the required
memory needed for state storage to 16.43% of the original. This saves about 283 MB of memory for state storage and 75 MB for the stack, making it possible to verify the specification on the given system.

Example:

```
$spin -a Commit.pml
$gcc -o pan -D_POSIX_SOURCE -DMEMCNT=27 -DSAFETY -DNOCLAIM -DNOFAIR -DCOLLAPSE pan.c
$time ./pan -m800000 -w19 -c1

State-vector 132 byte, depth reached 718303, errors: 0
2.34512e+06 states, stored
3.39051e+06 states, matched
5.73562e+06 transitions (= stored+matched)
2.00228e+07 atomic steps
hash conflicts: 8.25798e+06 (resolved)

Stats on memory usage (in Megabytes):
337.697 equivalent memory usage for states
  (stored*(State-vector + overhead))
  compressed State-vector=12 byte+12 byte overhead
55.471 memory usage for states (compression: 16.43%)
2.097 +memory used for hash-table (-w19)
22.400 +memory used for DFS stack (-m800000)
5.319 +memory used for other data structures
85.220 =total actual memory usage
```

Notice also that the number of states stored is much less than the number of states matched. Stored states are states encountered for the first time during the verification run. Matched states are states encountered during a verification run that have already been visited and entered into the state table. The difference in numbers results from the use of atomic sequences in the specification to reduce complexity. Without the use of atomic sequences, the number of unique states in the system would be much greater and would increase the memory requirements for verifying all properties of the system.

This specification requires the use of end-state labels to verify safety properties. This is due to the fact that the coordinator and workers are continually trying to receive messages. Therefore, end-state labels are placed at the main loops.
(lines 78, 280, and 351) for each of these processes. This prevents the verifier from falsely reporting these states as invalid end-states.

**Example.**

```bash
$spin -a Commit.pml
$gcc -o pan -D_POSIX_SOURCE -DMEMCNT=28 -DNP -DNOCLAIM
   -DNOFAIR -DCOLLAPSE pan.c
$time ./pan -m800000 -w19 -1 -cl

State-vector 136 byte, depth reached 761108, errors: 0
2.87517e+06 states, stored (3.48185e+06 visited)
4.41311e+06 states, matched
7.89496e+06 transitions (= visited+matched)
2.59671e+07 atomic steps
hash conflicts: 1.11732e+07 (resolved)

Stats on memory usage (in Megabytes):
425.526 equivalent memory usage for states
   (stored*(State-vector + overhead))
    compressed State-vector=12 byte+12 byte overhead
70.410 memory usage for states (compression: 16.55%)
2.097 +memory used for hash-table (-w19)
22.400 +memory used for DFS stack (-m800000)
4.410 +memory used for other data structures
99.249 =total actual memory usage
```

A verification for non-progress is illustrated in the above example; this type of verification searches for starvation loops. The placement of progress-state labels should be deferred until the verifier exposes a non-progress cycle. This avoids inadvertently placing one of these labels in the path of an undesirable cycle. Placing progress-state labels is more difficult than placing end-state labels because progress-state labels involve cyclic sequences of the global system execution, whereas deadlock simply presents a set of blocked statements for each of the processes in the system.

One progress-state label is placed in the *Client* process definition at line 31. This signifies that a loop that encompasses the *unless* statement in the client process is making progress. This should be obvious, because the client is keeping the system going by generating transaction requests. Any other cyclic
sequences of states occurring infinitely often that involve just the coordinator and workers would be flagged as an error.

A search for non-progress is actually modeled as a temporal claim under the hood. For a complete explanation on searching for non-progress cycles, see Chapter 5. The increase in memory requirements for a verification run of this type is modest—requiring only 15 MB of additional memory.

The cost of the compression algorithm used above is minimal, given that the alternative is the inability to perform verifications at all. The run-time for verifying safety properties was approximately 16 minutes, and the run-time for verifying liveness was just under 27 minutes. Holzmann claims that compression increases the verification run by a factor of 2-3. Due to memory constraints, however, this could not be confirmed for this specification.
The greatest difficulty with learning PROMELA/SPIN is the lack of a single up-to-date source of information. Holzmann's book, *Design and Validation of Computer Protocols*, was published in 1991 and has not been updated since. The problem is compounded by the numerous changes made to PROMELA/SPIN since the book's publication.

The documentation distributed with the source code attempts to track the changes made to the language and the tool. The documentation, however, is scattered throughout many files and is often contradictory. A single comprehensive source of information that is updated regularly would help tremendously in learning PROMELA/SPIN faster and using it more effectively.

After sifting through the documentation and papers to understand the latest release, it is not difficult to set up and use. Over the course of this thesis, SPIN was installed and used on four different platforms: a PC running Microsoft Windows95; a DEC Alpha running Digital Unix; a Sun workstation running SunOS; and an HP workstation running HP-UX. The installations were simple, requiring little, if any, modification to the source code and install scripts.

One can quickly learn PROMELA well enough to start creating simple specifications within a short time. For more complex or larger specifications, however, experience with the language is needed. This is especially evident
when verification runs start exhausting available memory. When this occurs, it is imperative to use the best possible constructs of the language that incur the least possible cost with respect to the state space.

Including more data types and functions in PROMELA would make the modeling process easier, however, SPIN's ability to analyze the resulting specifications would be compromised: Increasing modeling power usually decreases analytical power, and vice versa.

PROMELA/SPIN successfully balances the trade-off between modeling power and analytical power, making it a useful automated verification tool. This is exemplified by no explicit representation of time. The addition of real-time would increase state space by at least an order of magnitude for most systems. Consequently, PROMELA/SPIN would no longer be a viable formal verification method because it would only be able to verify very simple systems.

Mastering the ability to create useful specifications is not only a matter of understanding PROMELA syntax and semantics; it also requires a deep understanding of the subject matter that is to be modeled. A lack of this understanding hinders one's ability to create specifications at different levels of abstraction that accurately represent the design.

Critics maintain that system designers are already overburdened by the complexity of the problem domain, and that too much knowledge of formal methods is required to use them effectively. This view of formal methods is partly responsible for their limited acceptance. While it is true that knowledge of PROMELA and SPIN must be acquired to use them effectively, by removing the guesswork from the design process, systems can be designed more quickly and reliably.
For experienced users, SPIN is easy to use from the command line or through the graphical user interface XSPIN. Inexperienced users will benefit more from using XSPIN, and they can do so with relatively few problems. Most problems inexperienced users encounter result from using compilation directives and run-time options. Neither SPIN nor XSPIN do a great job of optimizing these for performance; this is left to the user. Other than these minor issues, the simulator and verifier perform well.

This thesis illustrates PROMELA/SPIN's ability to model three different types of systems—hardware circuits, transaction protocols, and communications protocols. PROMELA/SPIN is best suited for the development of communications protocols, the area for which it is designed. Chapter 3 shows that its usefulness for hardware verification is questionable.

The Commit Protocol was successfully verified (see Chapter 6) using PROMELA/SPIN, but not without difficulty. This is partly due to the nature of the protocol: It requires transactions to operate. A communications protocol (see Appendix I) can be modeled as a non-interactive continuous system, whereas a transaction protocol must have an oracle generate inputs and monitor output. Even using partial-order reduction methods, the system will inevitably grow with the size of the input space.

Extensions to PROMELA/SPIN make it useful in a greater number of application domains. Researchers have taken two approaches to extending PROMELA/SPIN. The first approach targets a specific set of related concurrent/distributed problems. Both the language and the tool are then optimized to address the issues in the chosen set. Two examples of this approach are PROMELA R/T for real-time systems, and PROMELA C/S for client-server systems.
The second approach to extending PROMELA/SPIN attempts to diminish the effect of state space explosion. The size of the state space is the limiting factor in model checking, and many methods of dealing with it have been proposed. One such method that has proven to work well is partial-order reduction; this is in the version of PROMELA/SPIN used in this thesis. Another method is state space caching, which reduces memory requirements by not requiring that all visited states be stored in memory.

While both approaches have advantages, the second approach has few disadvantages. Reducing the state space explosion effect is generally applicable, and may be used in collaboration with any specialized implementation of PROMELA/SPIN. Constraining PROMELA/SPIN to a specific set of problems requires implementing and maintaining each specialized version, which would be an overwhelming task.

There are a number of possible projects that can build on this thesis’ results. As more extensions of PROMELA/SPIN become available, they can be compared and evaluated. Temporal model checking, which is already widely used commercially for hardware verification, can be investigated. Another topic would be the translation of verified PROMELA models into a language such as C++ or Java.
8 REFERENCES


9 APPENDICIES

A. PROMELA Reserved Words/Labels .......................................................... 101
B. PROMELA Operators ........................................................................... 108
C. SPIN Simulation Options ................................................................. 109
D. SPIN LTL Operators ......................................................................... 110
E. SPIN Compilation Directives ............................................................... 111
F. SPIN Verification Options ................................................................. 114
G. D-Latch Specification ........................................................................ 115
H. Alternating-Bit Protocol Specification (Simulation Model) .............. 117
I. Alternating-Bit Protocol Specification (Verification Model) .......... 120
J. Commit Protocol Specification ............................................................ 122
A. PROMELA RESERVED WORDS

active  If this word precedes a proctype definition, then upon initialization of the system one or more of the corresponding processes will be instantiated. This is the only way to make active any other processes than the init process in the initial system state. All formal parameters of these processes will be initialized to zero. Two examples follow: the first instantiates a single process, while the second instantiates two.

Examples:  

active proctype SingleProcess() { ... }  
active [2] proctype TwoProcesses() { ... }

assert  Correctness claim that checks the validity of its argument during execution. If the argument evaluates to false (or zero), then it is reported as an error. The argument can be any expression that can be evaluated to a single value.

Example:  

assert( something == somethingElse )

atomic  A non-deterministic sequence of statements that are executed as one indivisible block. The block (sequence) of statements can be regarded as a kind of primitive because they cannot be interleaved with any other processes. This means that no other process can execute statements from the moment that the atomic sequence begins to the moment that it completes. The exception to this, however, is if one of the statements in the sequence blocks—control will be turned over to another process until that statement becomes executable.

Example:  

atomic( /*sequence of statements goes here*/ )

bit  A basic data type that is used to identify a variable representing one bit of information which can have a value of 0 or 1.

Example:  

bit oneBitOfInformation

bool  A basic data type that is used to identify a variable representing one bit of information which can have a value of 0 or 1—meaning false or true, respectively.

Example:  

bool oneBitOfInformation
break Is a loop termination statement used to jump out of a loop. When executed, it signals a jump to the end label of the current loop.

Example: break

byte A basic data type that is used to identify a variable representing eight bits of information which can have a value between 0 and 255, inclusive.

Example: byte oneByteOfInformation

chan Used to indicate that a variable is a channel. A channel is declared using both an array and a set notation. The brackets (array notation) encompass a positive integer that specifies the length of the channel. A channel that is declared with a length of '0' cannot store any messages and is only used for rendezvous communication. The braces (set notation) encompass the type(s) that are passed as a single message. If there is more than one type identifier in the braces, then each message will contain one value pertaining to each identifier. The examples below illustrate a rendezvous channel with messages of type bit and a channel of length ten messages where each message contains two integers.

Example: chan rendezvous = [0] of { bit }
          chan twoInts = [10] of { int, int }

d_step Indicates a sequence of statements that are executed as one indivisible block—much like atomic sequences. The block of statements in a d_step sequence, however, must be deterministic. This means that the sequence of statements cannot contain any jumps to destinations which are outside of the d_step sequence and that no statement other than the first may block. Any non-determinism encountered is resolved by taking the path with the first true guard.

Example: d_step(/*sequence of statements goes here*/)
do (od)  Represent the beginning and end, respectively, of loop structures. Each statement (or sequence of statements) in the loop that is preceded by a double colon represents a path that can be non-deterministically taken with each iteration. This provides a built-in case-type statement in the loop structure. Exiting a loop can be done by either “jumping” out of it, or by using a break statement.

Example:  
```
  do  /* do something */
     /* do something else */
     break
  od
```

else  Can be used as a guard in either a do construct or an if construct. It signifies in both cases a sequence that can be taken only when no other sequences in that (do or if) construct are executable.

Example:  
```
  . .
  :: else - > /*remainder of statement sequence*/
  . .
```

empty  A function performed on a channel which, consequently, is passed as its only argument. If there are no messages in the channel at the time of execution then the function evaluates to the boolean value of true, otherwise it will evaluate to false. It is not side-effect free.

Example:  `empty( myChannel )`

enabled  A function that takes as its only argument a process id (integer). If the process corresponding to the given process id can perform an executable operation at the time that this function is executed, then the function evaluates to the boolean value of true, otherwise it will evaluate to false. These statements are restricted to reside in never claims only.

Example:  `enabled( myProcessID )`

eval  A function that takes a variable name as its argument, and casts its current value to a constant. This way, the value of the variable can be used to match receive operations.

Example:  `eval( aVariable )`
full  A function performed on a channel which, consequently, is passed as its only argument. If the number of messages in the channel at the time of execution is equal to the predefined maximum value then the function evaluates to the boolean value of true, otherwise it will evaluate to false. It is not side-effect free.
   Example:  full( myChannel )

goto  The infamous goto--it is an unconditional jump.
   Example:  goto aDestination

hidden  Declares scratch variables that are write-only. This means that they do not carry meaningful values that can be used.
   Example:  hidden byte myScratchByte

if (fi) Represent the beginning and end, respectively, of selection structures. Each statement (or sequence of statements) in the structure that is preceded by a double colon represents a path that can be non-deterministically taken. A path can only be taken if the first statement, called a guard, evaluates to true (non-zero).
   Example:  if
             :: /* do something */
             :: /* do something else */
             fi

init  Used something like main in C and can only be used once. It is the only process to be instantiated (except for those labeled with the keyword active) in the initial system state.
   Example:  init( run thisProcess( ); run thatProcess( ) )

int  A basic data type that is used to identify a variable representing thirty-two bits of information which can have a value between $2^{31}$ and $2^{31} - 1$, inclusive.
   Example:  int signedNumber

len  A function that takes the label of a channel as its only argument and returns the number of messages the given channel currently contains. This function is not side-effect free.
   Example:  len( criticalChannel )
**mtype**  
Used to declare variables of the basic data type, **mtype**, which stands for message type. It is an unsigned quantity having a value between 0 and 255, inclusive. It is used to abstract from specific variable names so that they can be interpreted symbolically.

*Example:*  
\[
\text{mtype} = \{ \text{ack, nack} \};  
\text{chan response} = [1] \text{ of } \text{mtype} 
\]

**nempty**  
A function performed on a channel which, consequently, is passed as its only argument. If there are messages in the channel at the time of execution then the function evaluates to the boolean value of true, otherwise it will evaluate to false. This function is side-effect free and will not violate exclusive access assertions on channels.

*Example:*  
\[
\text{nempty( myChannel )} 
\]

**never**  
A temporal claim that is deemed to be impossible within the specified system. This means that the temporal claim is used to specify any illegal behavior in the orderings of the properties of the states of the system. A never claim can only contain propositions (conditional expressions) that will not inadvertently change the state of the system.

*Example:*  
\[
\text{never( /* behavior to look for */ )} 
\]

**nfull**  
A function performed on a channel. If the number of messages in the channel at the time of execution is less than the predefined maximum value then the function evaluates to the boolean value of true, otherwise it will evaluate to false. It is side-effect free and will not violate exclusive access assertions on channels.

*Example:*  
\[
\text{nfull( myChannel )} 
\]

**of**  
This word is used in the definition of channels—see **chan**.

**pc_value**  
A function that takes as its argument a process id and returns the current state of the process. The state number returned by the function corresponds to the line number of the PROMELA source code. Also, these statements are restricted to reside inside of **never** claims.

*Example:*  
\[
\text{pc_value( myProcessID )} 
\]
printf  A statement used to display output just like it is used in C.
    Example:  printf( "The result is: %d\n", result )

priority  Used when instantiating processes to specify the priority of the process. The default priority is 1 and a higher number indicates that it is N times more likely to execute than a process with a priority of 1. Can be used with active process also.
    Example:  proctype MyProcess() { /*define process here* /
        run MyProcess() priority N
        active proctype MyProc() { } priority N

proctype  Used in the declaration of new process types.
    Example:  proctype MyProcess() { /* define process here */ }

provided  Used in the declaration of new process types to specify an enabling condition. The enabling condition must be side-effect free.
    Example:  proctype MyProcess() { } provided(expression)

run  Used to spawn new processes.
    Example:  run MyProcess() 

short  A basic data type that is used to identify a variable representing sixteen bits of information which can have a value between \(-2^{15}\) and \(2^{15}-1\), inclusive.
    Example:  short signedNumber

skip  A statement that has no effect on the current state of the system. It is only used as a place holder.
    Example:  skip

timeout  Guards against a deadlocked system state. If there is a process in the system somewhere that is currently at a timeout statement, then the timeout will become executable and execute if no other statement in the system can. This statement is especially useful for watchdog timers.
    Example:  timeout \(-\longrightarrow\) * reset something */

typedef  Declares user-defined data types—similar to struct in C.
    Example:  typedef myDataType{ bool field1; int field2 }
unless  An escape sequence that can change control from a statement (or set of statements) to another statement (or set of statements). In the example below the process would loop forever. However, if the guard C of the escape sequence, became executable while either A or B were executing, then control of the process would change and C would execute.

Example:  do  
:: A -> B
od
unless { C -> D }

xr  The assertion made using this keyword states that the process that the statement is found in is the only process that will read from that channel. This is necessary for partial order reduction.

Example:  xr myChannel

xs  The assertion made using this keyword states that the process that the statement is found in is the only process that will write to that channel. This is necessary for partial order reductions.

Example:  xs myChannel

This is a predefined scratch (write-only) variable.
B. PROMELA OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater-than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater-than or Equal-to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less-than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less-than or Equal-to</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not Equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>Complement</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift Right</td>
</tr>
<tr>
<td>?</td>
<td>Receive</td>
</tr>
<tr>
<td>??</td>
<td>Random Receive</td>
</tr>
<tr>
<td>!</td>
<td>Send</td>
</tr>
<tr>
<td>!!</td>
<td>Sorted Send</td>
</tr>
</tbody>
</table>
## C. SPIN SIMULATION OPTIONS

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Indicates an interactive simulation run.</td>
</tr>
<tr>
<td>t</td>
<td>Indicates a guided simulation run using a generated simulation trail.</td>
</tr>
<tr>
<td>jN</td>
<td>Will skip the first N steps of a guided simulation run.</td>
</tr>
<tr>
<td>nN</td>
<td>Sets the seed of the random number generator to N.</td>
</tr>
<tr>
<td>m</td>
<td>Loose the messages sent to a full queue.</td>
</tr>
<tr>
<td>l</td>
<td>Print the values of all local variables.</td>
</tr>
<tr>
<td>g</td>
<td>Print the values of all global variables.</td>
</tr>
<tr>
<td>s</td>
<td>Print all send operations.</td>
</tr>
<tr>
<td>r</td>
<td>Print all receive operations.</td>
</tr>
<tr>
<td>p</td>
<td>Print all statements executed.</td>
</tr>
<tr>
<td>v</td>
<td>Print all warning information available.</td>
</tr>
<tr>
<td>d</td>
<td>Print a symbol table containing all initial system information.</td>
</tr>
<tr>
<td>V</td>
<td>Print the current version of the tool being used.</td>
</tr>
<tr>
<td>f</td>
<td>Translate an LTL formula into a never claim.</td>
</tr>
<tr>
<td>a</td>
<td>Generate a verifier called pan.c.</td>
</tr>
</tbody>
</table>
D. SPIN LTL OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \land )</td>
<td>Logical AND</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td>( \lor )</td>
<td>Logical OR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Logical Negation</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Logical Implication</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>Logical Equivalence</td>
</tr>
<tr>
<td>[ ]</td>
<td>Always</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Eventually</td>
</tr>
<tr>
<td>U</td>
<td>Until</td>
</tr>
<tr>
<td>V</td>
<td>Dual of Until</td>
</tr>
</tbody>
</table>
# E. SPIN COMPILATION DIRECTIVES

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITSTATE</td>
<td>Compiles for the <em>Supertrace</em> (bit-state space) algorithm. Used for systems that are too large for exhaustive verification.</td>
</tr>
<tr>
<td>CHECK</td>
<td>Provides information to be printed out during verification for debugging. Gives less information than VERBOSE.</td>
</tr>
<tr>
<td>COLLAPSE</td>
<td>Can compress the state vector by 90%. Used to save memory during verifications, but increases the run-time by 2 or 3 times.</td>
</tr>
<tr>
<td>COVEST</td>
<td>Provides an estimate of coverage for a bit-state space analysis of the system.</td>
</tr>
<tr>
<td>CTL</td>
<td>Will only allow for partial-order reduction that is consistent with branching-time temporal logics.</td>
</tr>
<tr>
<td>HYBRID_HASH</td>
<td>Saves one byte for every state vector when the state vector is one byte longer than a multiple of four.</td>
</tr>
<tr>
<td>MEMCNT</td>
<td>Sets the upper-bound for the maximum allowable memory to be used for a verification run. This constrains the tool to RAM.</td>
</tr>
<tr>
<td>NFAIR</td>
<td>Allocates memory for enforcing weak fairness. It is only used when prompted by the tool.</td>
</tr>
<tr>
<td>NIBIS</td>
<td>Used for extra optimization in conjunction with partial-order reduction.</td>
</tr>
<tr>
<td>NOBOUNDCHECK</td>
<td>Will not check for violations of array bounds. Used to perform faster verification runs.</td>
</tr>
<tr>
<td>NOCLAIM</td>
<td>Disables the use of the <em>never</em> claim for the verification run.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>NOCOMP</td>
<td>Will not compress the state vector. Used to perform faster verification runs.</td>
</tr>
<tr>
<td>NOCOND</td>
<td>Disables conditional safety rules for faster verification runs.</td>
</tr>
<tr>
<td>NOFAIR</td>
<td>Disables weak-fairness for faster verification runs and less memory usage.</td>
</tr>
<tr>
<td>NOREDUCE</td>
<td>Disables partial-order reduction for full state space exploration.</td>
</tr>
<tr>
<td>NOSTUTTER</td>
<td>Disables stuttering of the final state of the never claim, however, it does change the semantics.</td>
</tr>
<tr>
<td>NOVSZ</td>
<td>Removes the length field, which is four bytes, from the state vector to save memory. It may invalidate liveness checks.</td>
</tr>
<tr>
<td>NP</td>
<td>Enables a check for non-progress cycles during verification.</td>
</tr>
<tr>
<td>PEG</td>
<td>Enables complexity profiling during verification.</td>
</tr>
<tr>
<td>PRINTF</td>
<td>Enables printf statements during verification.</td>
</tr>
<tr>
<td>REACH</td>
<td>Can guarantee the absence of errors within the limit of the stack. A truncated search cannot always guarantee the absence errors due to not exploring previously visited states.</td>
</tr>
<tr>
<td>SAFETY</td>
<td>Can be used when not checking for liveness for faster verifications and less memory usage.</td>
</tr>
<tr>
<td>SDUMP</td>
<td>Produces an ascii file at the end of a verification run that contains the state vectors for all states.</td>
</tr>
<tr>
<td>SVDUMP</td>
<td>Produces a binary file at the end of a verification run that contains the state vectors for all states.</td>
</tr>
<tr>
<td>VAR_RANGES</td>
<td>Produces the ranges of all variables during a verification run. It is restricted compute values from 0 to 255.</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VECTORSZ</td>
<td>Allocates memory for the state vector.</td>
</tr>
<tr>
<td>VERBOSE</td>
<td>Provides information for debugging during verification.</td>
</tr>
<tr>
<td>XUSA FE</td>
<td>Disables validity checks of channel assertions when the check is too strict.</td>
</tr>
</tbody>
</table>
F. SPIN VERIFICATION OPTIONS

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Search for acceptance cycles.</td>
</tr>
<tr>
<td>b</td>
<td>Uses backward single-bit hashing.</td>
</tr>
<tr>
<td>cN</td>
<td>Halts a verification run upon reaching the N&lt;sup&gt;th&lt;/sup&gt; error.</td>
</tr>
<tr>
<td>d</td>
<td>Prints all of the information in the state tables.</td>
</tr>
<tr>
<td>e</td>
<td>Saves state trails for all encountered errors.</td>
</tr>
<tr>
<td>f</td>
<td>Enforces weak-fairness.</td>
</tr>
<tr>
<td>hN</td>
<td>Allows a user to choose a list of hash functions numbering 1 to 32.</td>
</tr>
<tr>
<td>i</td>
<td>Search for the shortest error trail.</td>
</tr>
<tr>
<td>I</td>
<td>Faster and approximate search for the shortest error trail.</td>
</tr>
<tr>
<td>l</td>
<td>Search for non-progress cycles.</td>
</tr>
<tr>
<td>mN</td>
<td>Sets the maximum search depth to N.</td>
</tr>
<tr>
<td>n</td>
<td>Disables the listing of unreachable states.</td>
</tr>
<tr>
<td>q</td>
<td>Enforces the requirement of empty channels in valid end-states.</td>
</tr>
<tr>
<td>s</td>
<td>Uses forward single-bit hashing.</td>
</tr>
<tr>
<td>V</td>
<td>Prints the current version of the tool being used.</td>
</tr>
<tr>
<td>wN</td>
<td>Sets the size of the hash table to $2^N$.</td>
</tr>
</tbody>
</table>
G. D-LATCH SPECIFICATION

/*
 * Model: D-Latch
 * File Name: DLatch.pml
 * Author: Mark Bezdany
 */
#define INVERTER(I, O) (O != (1-I)) -> O = 1-I
#define AND2(I1, I2, O) (O != (I1&&I2)) -> O = (I1&&I2)
#define OR3(I1,I2,I3,O) (O!=(I1||I2||I3)) -> O = (I1||I2||I3)
#define NAND2(I1, I2, O) (O != !(I1&&I2)) -> O = !(I1&&I2)
#define NAND3(I1,I2,I3,O) (O!= !(I1&&I2&&I3)).->O=!(I1&&I2&&I3)

bit d, en, q, qOld, newInput;

proctype DLatchValid() {
    bit 11, 12, 13, 14;
    do
:: if
    :: INVERTER(en, 11)
    :: AND2(d, q, 12)
    :: AND2(d, en, 13)
    :: AND2(11, q, 14)
    :: OR3(12, 13, 14, q)
    fi;
    newInput = 0
    od
}

proctype DLatchInvalid() {
    bit 11, 12, 13, 14;
    do
:: if
    :: INVERTER(en, 11);
    :: NAND2(d, q, 12);
    :: NAND2(d, en, 13);
    :: NAND2(11, q, 14);
    :: NAND3(12, 13, 14, q);
    fi;
    newInput = 0
    od
}
proctype Stimulus() {
    do :: timeout -> atomic { newInput = 1;
        qOld = q;
        if :: d = 1 - d
            :: en = 1 - en
        fi }
    od
}

init {
    newInput = 1;
    /* Run with one of the two following atomic
        statements commented out. */
    /* atomic { run Stimulus(); run DLatchInvalid() } */
    atomic { run Stimulus(); run DLatchValid() }
}

never {
    Top:
        /* If there has been no new input and the output
            has changed, then break out of this loop. */
    do :: ((newInput == 0) && (qOld != q)) -> break
        :: else -> skip
    od;

do
    /* Conditions haven't changed from previous loop. */
    :: ((newInput == 0) && (qOld != q))

    /* Still no new input, but output changed again. */
    :: ((newInput == 0) && (qOld == q)) -> break

    /* Input changed, so restart. */
    :: (newInput == 1) -> goto Top
od
}
H. ALTERNATING-BIT PROTOCOL SPECIFICATION
SIMULATION MODEL

/*
 * Model: Alternating-Bit Protocol (Simulation)
 * File Name: FullDuplexAB_S.pml
 * Author: Mark Bezdany
 */
#define STOP_AT 3
#define MAX_TIME 7
#define ERROR_RATE1 256
#define ERROR_RATE2 256

init {
  chan StoM = [2] of { bit, bit, byte };  
  chan MtoR = [2] of { bit, bit, byte }; 
  chan RtoM = [2] of { bit, bit, byte }; 
  chan MtoS = [2] of { bit, bit, byte }; 
  atomic {
    run SenderReceiver(MtoS, StoM, 1);  
    run Medium(StoM, MtoS, RtoM, MtoR);  
    run SenderReceiver(MtoR, RtoM, 0) 
  }
}

proctype Medium(chan from1, to1, from2, to2) {
  byte errorByte, tempByte; 
  bit tempBit1, tempBit2, looseMsg, badMsg; 
  errorByte = 1; 
  Top:
  do 
    :: skip -> 
    if 
      :: nempty(from1) -> from1?tempBit1,tempBit2,tempByte  
      :: empty(from1) -> goto Reset1 
    fi; 
    
    if 
      :: (looseMsg == 1) -> goto Reset2  
      :: (badMsg == 1) -> to2!tempBit1,tempBit2,0 
      :: else -> to2!tempBit1,tempBit2,tempByte 
    fi;
Reset1:
  looseMsg = 0; badMsg = 0;
  if
    :: (errorByte%ERROR_RATE1==0) -> looseMsg=1
    :: (errorByte%ERROR_RATE1==0) -> badMsg = 1
    :: else -> skip
  fi;

  if
    :: nempty(from2) -> from2?tempBit1,tempBit2,tempByte
    :: empty(from2) -> goto Reset2
  fi;

  if
    :: (tempByte == 255) -> goto StopMedium
    :: else -> skip
  fi;

  if
    :: (looseMsg == 1) -> goto Reset1
    :: (badMsg == 1) -> tol!tempBit1,tempBit2,0
    :: else -> tol!tempBit1,tempBit2,tempByte
  fi;

Reset2:
  looseMsg = 0; badMsg = 0;
  if
    :: (errorByte%ERROR_RATE2==0) -> looseMsg=1
    :: (errorByte%ERROR_RATE2==0) -> badMsg = 1
    :: else -> skip
  fi;

  errorByte++
  od;

StopMedium:
  skip
proctype SenderReceiver(chan in, out; bit first) {
    byte clock, lastMsgln, msgIn, msgOut;
    bit controlIn, ackIn, controlOut, ackOut;
    atomic {
        if
            : (first == 1) -> ackOut = 1; msgOut = 1
            : else -> controlOut = 1; goto Receive
        fi
    );
    Send:
        atomic(out!controlOut,ackOut,msgOut; clock = 0);
    Receive:
        do
            : nempty(in) -> in?controlIn,ackIn,msgIn; break
            : empty(in) ->
                if
                    : (clock >= MAX_TIME) -> goto Send
                    : else -> clock++
                fi;
            od;
        if
            : (msgIn==lastMsgIn) -> goto Receive
            : else -> lastMsgIn = msgIn
        fi;
        if
            : ((msgIn > 0) && (msgIn != 255)) -> ackOut = controlIn
            : (msgIn == 255) -> out!0,0,255; goto Stop
            : else -> goto Send
        fi;
        if
            : (ackIn == controlOut) -> controlOut = 1 - controlOut;
            msgOut++
        : else -> goto Send
        fi;
        if
            : (msgOut >= STOP_AT) -> out!0,0,255
        : else -> goto Send
        fi;
    Stop:
        skip
    }
}
I. ALTERNATING-BIT PROTOCOL SPECIFICATION
VERIFICATION MODEL

/*
 * Model: Alternating-Bit Protocol (Verification)
 * File Name: FullDuplexAB_V.pml
 * Author: Mark Bezdany
 */
#define MAX_TIME 7
mtype = {GOOD_PACKET, BAD_PACKET};

proctype SenderReceiver(chan in, out; bit first) {
    byte clock, msgIn;
    bit controlIn, ackIn, controlOut, ackOut;
    xr in; xs out;
    if
        .: (first == 1) -> ackOut = 1
        :: else -> controlOut = 1; goto Receive
    fi;

    Send:
    atomic {
        if
            :: out!controlOut,ackOut,GOOD_PACKET; clock=0
            :: out!controlOut,ackOut,BAD_PACKET; clock=0
            :: skip
        fi;
    };

    Receive:
    do
        .: (clock >= MAX_TIME) -> goto Send
        :: else -> clock++
    od
    unless { nempty(in) -> in?controlIn,ackIn,msgIn };

    if
        :: (ackIn==controlOut && msgIn==GOOD_PACKET) ->
            controlOut = 1 - controlOut; ackOut = controlIn
        :: (ackIn!=controlOut && msgIn==GOOD_PACKET) ->
            ackOut = controlIn
        :: else -> skip
    fi;
    goto Send
}

120
init {
    chan StoR = [1] of { bit, bit, mtype };
    chan RtoS = [1] of { bit, bit, mtype };
    atomic {
        run SenderReceiver(RtoS, StoR, 1);
        run SenderReceiver(StoR, RtoS, 0);
    }
}
/* Model: Two-Phase Commit Protocol */

#define MAX_TRANS 2
#define MAX_DATA 2
#define MESSAGE_SIZE 7
#define LOCK_SIZE 3
#define TID_POS 6
#define CH_SZ 1

mtype = { NewTrans, CanCommit, Can, Cant, Abort, DoCommit, HaveCommitted, Committed, Aborted };
typedef dataEntry { bit lockTransID[LOCK_SIZE] };
typedef message { bit dataTransID[MESSAGE_SIZE] };
active proctype Client() {
    xr C0ToClient; xs ClientToC0;
    bit transOut1; bit transOut2;
    message trans;
    mtype messType;

    TransLoop:
    progress_Cl:
    do
        :: atomic ( ((transOut1 == 0) || (transOut2 == 0)) ->
            if
                :: trans.dataTransID[0] = 1 - trans.dataTransID[0]
                fi;

        if
            ((trans.dataTransID[0]+trans.dataTransID[1])>=1) &
            ((trans.dataTransID[4]+trans.dataTransID[5])>=1)) ->
                if
                    :: (transOut1 == 0) ->
                    trans.dataTransID[TID_POS] = 0;
                    transOut1 = 1
                    :: (transOut2 == 0) ->
                    trans.dataTransID[TID_POS] = 1;
                    transOut2 = 1
                    fi;
                ClientToC0!NewTrans,trans
            else -> skip
                fi
        od
    unless
        { nempty(C0ToClient) ->
            if
                :: C0ToClient?messType,0 -> transOut1 = 0
                :: C0ToClient?messType,1 -> transOut2 = 0
                fi
        }
    goto TransLoop;
}
active proctype Coordinator() {
  xr ClientToC0;  xs C0ToClient;
  xr W0toC0;       xs C0toW0;
  xr WltoC0;       xs C0toWl;
dataEntry dataTable[MAX_DATA];
  bit phaseT0;     bit phaseTl; bit votes[8]; bit abort;
  message trans;

  CoordStart:
  end_C0:
  { ( atomic { ClientToC0?NewTrans,trans -> goto C0Handlel } )
  unless
  { (nempty(W0toC0) || nempty(WltoC0)) ->
    if
    :: W0toC0?Can,trans ->
      if
      :: (trans.dataTransID[TID_POS] == 0) -> votes[0] = 1
      :: (trans.dataTransID[TID_POS] == 1) -> votes[4] = 1
      fi;
    :: WltoC0?Can,trans ->
      if
      :: (trans.dataTransID[TID_POS] == 0) -> votes[1] = 1
      :: (trans.dataTransID[TID_POS] == 1) -> votes[5] = 1
      fi;
    :: W0toC0?Cant,trans ->
      if
      :: (trans.dataTransID[TID_POS] == 0) ->
        votes[0] = 1; votes[2] = 1
      :: (trans.dataTransID[TID_POS] == 1) ->
      fi;
    :: WltoC0?Cant,trans ->
      if
      :: (trans.dataTransID[TID_POS] == 0) ->
        votes[1] = 1; votes[3] = 1
      :: (trans.dataTransID[TID_POS] == 1) ->
      fi;
    :: W0toC0?HaveCommited,trans ->
      if
      :: (trans.dataTransID[TID_POS] == 0) -> votes[0] = 1
      :: (trans.dataTransID[TID_POS] == 1) -> votes[4] = 1
      fi;
    :: WltoC0?HaveCommited,trans ->
      if
      :: (trans.dataTransID[TID_POS] == 0) -> votes[1] = 1
      :: (trans.dataTransID[TID_POS] == 1) -> votes[5] = 1
      fi;
    goto C0Handle2
  };
goto CoordStart;
C0Handle1:
   atomic {
      if
         :: ((trans.dataTransID[0] == 1) &&
               (dataTable[0].lockTransID[0] == 1)) ->
            abort = 1
         :: ((trans.dataTransID[1] == 1) &&
               (dataTable[1].lockTransID[0] == 1)) ->
            abort = 1
         :: else -> skip
      fi;
      if
         :: (abort == 0) ->
            if
               :: (trans.dataTransID[0] == 1) ->
                  assert( dataTable[0].lockTransID[0] == 0 );
                  dataTable[0].lockTransID[0] = 1;
                  dataTable[0].lockTransID[
                     (trans.dataTransID[TID_POS] + 1)] = 1
               :: else -> skip
            fi;
            if
               :: (trans.dataTransID[1] == 1) ->
                  assert( dataTable[1].lockTransID[0] == 0 );
                  dataTable[1].lockTransID[0] = 1;
                  dataTable[1].lockTransID[
                     (trans.dataTransID[TID_POS] + 1)] = 1
               :: else -> skip
            fi;
            C0toWO!CanCommit,trans;
            C0toWl!CanCommit,trans
         :: else ->
            abort = trans.dataTransID[TID_POS];
            C0ToClient!Aborted,abort;
            abort = 0
            fi;
      fi;
   };
   goto CoordStart;
C0Handle2:

    atomic {
        if
            :: ((trans.dataTransID[TID_POS] == 0) && (phaseT0 == 0)) ->
                if
                    :: ((votes[0] == 1) && (votes[1] == 1)) ->
                        if
                            :: ((votes[2] == 1) && (votes[3] == 0)) ->
                                votes[0] = 0; votes[1] = 0; votes[2] = 0; votes[3] = 0;
                                C0toW0!Abort,trans;
                                C0toClient!Aborted,abort;
                                abort = 0
                            :: ((votes[2] == 0) && (votes[3] == 1)) ->
                                votes[0] = 0; votes[1] = 0; votes[2] = 0; votes[3] = 0;
                                C0toW0!Abort,trans;
                                abort = trans.dataTransID[TID_POS];
                                C0toClient!Aborted,abort;
                                abort = 0
                        :: ((votes[2] == 1) && (votes[3] == 1)) ->
                            votes[0] = 0; votes[1] = 0; votes[2] = 0; votes[3] = 0;
                            abort = trans.dataTransID[TID_POS];
                            C0toClient!Aborted,abort;
                            abort = 0
                        :: ((votes[2] == 0) && (votes[3] == 0)) ->
                            votes[0] = 0; votes[1] = 0; votes[2] = 0; votes[3] = 0;
                            phaseT0 = 1;
                            C0toW0!DoCommit,trans;
                            C0toW1!DoCommit,trans
                        fi;
                    :: (dataTable[0].lockTransID[0] == 1) ->
                        dataTable[0].lockTransID[0] = 0;
                        dataTable[0].lockTransID[1] = 0;
                        dataTable[0].lockTransID[2] = 0;
                    :: else -> skip
                fi;
            :: else -> skip
        fi;
((trans.dataTransID[TID_POS] == 0) && (phaseT0 == 1)) ->
if
  ((votes[0] == 1) && (votes[1] == 1)) ->
  votes[0]=0; votes[1]=0; votes[2]=0; votes[3]=0;
  phaseT0 = 0; abort = 0;
  abort = trans.dataTransID[TID_POS];
  COToClient!Commit, abort
  else -> skip
fi
((trans.dataTransID[TID_POS] == 1) && (phaseT1 == 0)) ->
if
  if
    ((votes[6] == 1) && (votes[7] == 0)) ->
      COToW1!Abort, trans;
      abort = trans.dataTransID[TID_POS];
      COToClient!Aborted, abort;
      abort = 0
    else
      ((votes[5] == 0) && (votes[7] == 1)) ->
      COToW0!Abort, trans;
      abort = trans.dataTransID[TID_POS];
      COToClient!Aborted, abort;
      abort = 0
    fi
  else
    ((votes[6] == 1) && (votes[7] == 1)) ->
      COToClient!Aborted, abort;
      abort = 0
  fi
  ((votes[6] == 1) && (votes[7] == 0)) ->
    COToW0!DoCommit, trans;
    COToW1!DoCommit, trans
fi;
if
  (dataTable[0].lockTransID[2] == 1) ->
    dataTable[0].lockTransID[0] = 0;
    dataTable[0].lockTransID[1] = 0;
    dataTable[0].lockTransID[2] = 0;
  else -> skip
fi;
if
  (dataTable[1].lockTransID[2] == 1) ->
    dataTable[1].lockTransID[0] = 0;
    dataTable[1].lockTransID[1] = 0;
    dataTable[1].lockTransID[2] = 0;
  else -> skip
fi;
fi
:: ((trans.dataTransID[TID_POS] == 1) && (phaseT1 == 1)) ->
if
:: ((votes[4] == 1) && (votes[5] == 1)) ->
phaseT1 = 0;
abort = trans.dataTransID[TID_POS];
C0ToClient!Commited, abort;
abort = 0
:: else -> skip
fi
fi;
}
}
goto CoordStart
}

active proctype W0() {
  xr C0toW0; xs W0toC0;
dataEntry dataTable[MAX_DATA];
message trans;
bit abort;

WOStart:
end_W0:
{
if
:: C0toW0?CanCommit, trans -> goto W0Handle1
:: C0toW0?DoCommit, trans -> goto W0Handle2
:: C0toW0?Abort, trans -> abort = 1; goto W0Handle2
fi
}
}
goto W0Start;

W0Handle1:
atomic {
if
:: ((trans.dataTransID[2] == 1) &&
  (dataTable[0].lockTransID[0] == 1)) ->
  abort = 1
:: ((trans.dataTransID[3] == 1) &&
  (dataTable[1].lockTransID[0] == 1)) ->
  abort = 1
:: else -> skip
fi;
if (abort == 0) ->
  if (trans.dataTransID[0 + MAX_DATA] == 1) ->
    assert (dataTable[0].lockTransID[0] == 0);
    dataTable[0].lockTransID[0] = 1;
    dataTable[0].lockTransID[
      (trans.dataTransID[TID_POS] + 1)] = 1
    :: else -> skip
  fi;
  if (trans.dataTransID[1 + MAX_DATA] == 1) ->
    assert (dataTable[1].lockTransID[0] == 0);
    dataTable[1].lockTransID[0] = 1;
    dataTable[1].lockTransID[
      (trans.dataTransID[TID_POS] + 1)] = 1
    :: else -> skip
  fi;
  W0toC0: Can, trans
  .. else -> abort = 0; W0toC0: Cant, trans
  fi;
  goto W0Start;
}

W0Handle2:
  atomic {
    if (dataTable[0].lockTransID[
      (trans.dataTransID[TID_POS] + 1)] == 1) ->
      dataTable[0].lockTransID[0] = 0;
      dataTable[0].lockTransID[1] = 0;
      dataTable[0].lockTransID[2] = 0;
      :: else -> skip
    fi;
    if (dataTable[1].lockTransID[
      (trans.dataTransID[TID_POS] + 1)] == 1) ->
      dataTable[1].lockTransID[0] = 0;
      dataTable[1].lockTransID[1] = 0;
      dataTable[1].lockTransID[2] = 0;
      :: else -> skip
    fi;
    if (abort == 0) -> W0toC0: HaveCommitted, trans
    .. else -> skip
  fi;
  abort = 0
  goto W0Start
}
active proctype Wl() {
    xr C0toWl; xs WltoC0;
    dataEntry dataTable[MAX_DATA];
    message trans;
    bit abort;

    WlStart:
    end_Wl:
    {
        if :: C0toWl?CanCommit,trans -> goto WlHandle1
            .:: C0toWl?DoCommit,trans -> goto WlHandle2
            :: C0toWl?Abort,trans -> abort = 1; goto WlHandle2
            fi
        goto WlStart;
    }

    WlHandle1:
    atomic {
        atomic {
            if :: (trans.dataTransID[4] == 1) &&
                (dataTable[0].lockTransID[0] == 1)) ->
                abort = 1
            fi

            if :: (abort == 0) ->
                if :: (trans.dataTransID[4] == 1) ->
                    assert ( dataTable[0].lockTransID[0] == 0 );
                    dataTable[0].lockTransID[0] = 1;
                    dataTable[0].lockTransID[trans.dataTransID[TID_POS] + 1]) = 1
                fi

            fi

            if :: (trans.dataTransID[5] == 1) ->
                assert ( dataTable[1].lockTransID[0] == 0 );
                dataTable[1].lockTransID[0] = 1;
                dataTable[1].lockTransID[trans.dataTransID[TID_POS] + 1]) = 1
                fi

            fi
        fi
    }

    WlHandle2:
    atomic {
        if :: C0toWl?Can,trans -> goto WlHandle2
            else abort = 0; WltoC0!Can,trans
            fi
        goto WlStart;
    }
}
WlHandle2:
atomic {
  if
    :: (dataTable[0].lockTransID[
       (trans.dataTransID[TID_POS] + 1)] == 1) ->
      dataTable[0].lockTransID[0] = 0;
      dataTable[0].lockTransID[1] = 0;
      dataTable[0].lockTransID[2] = 0;
    :: else -> skip
    fi;
  if
    :: (dataTable[1].lockTransID[
       (trans.dataTransID[TID_POS] + 1)] == 1) ->
      dataTable[1].lockTransID[0] = 0;
      dataTable[1].lockTransID[1] = 0;
      dataTable[1].lockTransID[2] = 0;
    :: else -> skip
    fi;
  if
    :: (abort == 0) -> WltoC0!HaveCommited,trans
    :: else -> skip
    fi;
  abort = 0
};
  goto WlStart
}