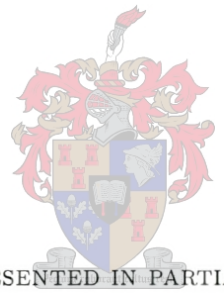
A comparison of two different model checking techniques

J.J.D. Bull



THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH.

Supervised by: Prof. P.J.A. de Villiers

December 2003

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Abstract

Model checking is a computer-aided verification technique that is used to verify properties about the formal description of a system automatically. This technique has been applied successfully to detect subtle errors in reactive systems. Such errors are extremely difficult to detect by using traditional testing techniques. The conventional method of applying model checking is to construct a model manually either before or after the implementation of a system. Constructing such a model requires time, skill and experience. An alternative method is to derive a model from an implementation automatically.

In this thesis two techniques of applying model checking to reactive systems are compared, both of which have problems as well as advantages. Two specific strategies are compared in the area of protocol development:

1. Structuring a protocol as a transition system, modelling the system, and then deriving an implementation from the model.
2. Automatically translating implementation code to a verifiable model.

Structuring a reactive system as a transition system makes it possible to verify the control flow of the system at implementation level—as opposed to verifying the control flow at abstract level. The result is a closer correspondence between implementation and specification (model). At the same time testing, which is restricted to small, independent code fragments that manipulate data, is simplified significantly.

The construction of a model often takes too long; therefore, verification results may no longer be applicable when they become available. To address this problem, the technique of automated model extraction was suggested. This technique aims to reduce the time required to construct a model by minimising manual input during model construction.

A transition system is a low-level formalism and direct execution through interpretation is

feasible. However, the overhead of interpretation is the major disadvantage of this technique. With automated model extraction there are disadvantages too. For example, differences between the implementation and specification languages—such as constructs present in the implementation language that cannot be expressed in the modelling language—make the development of an automated model extraction tool extremely difficult.

In conclusion, the two techniques are compared against a set of software development considerations. Since a specific technique is not always preferable, guidelines are proposed to help select the best approach in different circumstances.

Summary in Afrikaans

Modeltoetsing is 'n rekenaargebaseerde verifikasietegniek wat gebruik word om eienskappe rakende 'n formele spesifikasie van 'n stelsel te verifieer. Die tegniek is al suksesvol toegepas om subtiele foute in reaktiewe stelsels op te spoor. Sulke foute word uiters moeilik opgespoor as tradisionele toetsings tegnieke gebruik word. Tradisioneel word modeltoetsing toegepas deur 'n model te bou voor of na die implementasie van 'n stelsel. Om 'n model te bou verg tyd, vernuf en ervaring. 'n Alternatiewe metode is om outomaties 'n model van 'n implementasie af te lei.

In hierdie tesis word twee toepassingstegnieke van modeltoetsing vergelyk, waar beide tegnieke beskik oor voordele sowel as nadele. Twee strategieë word vergelyk in die gebied van protokol ontwikkeling:

1. Om 'n protokol as 'n oorgangsstelsel te struktureer, dit te moduleer en dan 'n implementasie van die model af te lei.
2. Om outomaties 'n verifieerbare model van 'n implementasie af te lei.

Om 'n reaktiewe stelsel as 'n oorgangsstelsel te struktureer maak dit moontlik om die kontrolevloei op implementasie vlak te verifieer—in teenstelling met verifikasie van kontrolevloei op 'n abstrakte vlak. Die resultaat is 'n nouer band wat bestaan tussen die implementasie en die spesifikasie. Terselfdetyd word toetsing, wat beperk word tot klein, onafhanklike kodesegmente wat data manupileer, beduidend vereenvoudig.

Die konstruksie van 'n model neem soms te lank; gevolglik, wanneer die verifikasieresultate beskikbaar word, is dit dalk nie meer toepaslik op die huidige weergawe van 'n implementasie nie. Om die probleem aan te spreek is 'n tegniek om modelle outomaties van implementasies af te lei, voorgestel. Die doel van die tegniek is om die tyd wat dit neem om 'n model te bou te verminder deur handtoevoer tot 'n minimum te beperk.

'n Oorgangsstelsel is 'n laevlak formalisme en direkte uitvoering deur interpretasie is wesenlik. Die oorhoofse koste van die interpreteerder is egter die grootste nadeel van die tegniek. Daar is ook nadele wat oorweeg moet word rakende die tegniek om outomaties modelle van implementasies af te lei. Byvoorbeeld, verskille tussen die implementasietaal en spesifikasietaal—soos byvoorbleed konstrukte wat in die implementasietaal gebruik word wat nie in die modeleringstaal voorgestel kan word nie—maak die ontwikkeling van 'n modelafleier uiters moeilik.

As gevolg word die twee tegnieke vergelyk teen 'n stel van programatuurontwikkelingsoorwegings. Omdat 'n spesifieke tegniek nie altyd voorkeur kan geniet nie, word riglyne voorgestel om te help met die keuse om die beste tegniek te kies in verskillende omstandighede.

Acknowledgements

I want to thank the following people:

- My supervisor, Pieter de Villiers, for guidance and encouragement in my studies.
- All the people who participated in countless debates regarding topics in this thesis, especially Frank, Jacques, Riaan and Willem.
- My family and friends for their support.

Contents

Chapter 1

Introduction

The production of reliable, concurrent software is a formidable task. One major problem is that race conditions that are caused by the timing of specific events are difficult to reproduce during testing. The problem has been studied extensively. For example, specialised testing techniques have been developed to allow a user to control the execution of concurrent processes. Another approach is to design concurrent programming languages that enable a compiler to check for commonly occurring programming errors. Yet another strategy involves the use of manual verification techniques and software tools that can detect subtle errors in designs.

One particularly successful verification technique is model checking. A model checker is a computer aided verification tool that can verify, without human assistance, that a formal specification (a model) of a system has certain desirable properties. The basic idea is to explore the reachable state space of a model to check a given correctness claim. To analyse a complex concurrent system, millions of states must be stored in main memory. Because a single state may occupy several hundreds of bytes, the amount of memory available on typical workstations becomes a limiting factor. Fortunately, several techniques have been developed to reduce the memory requirements. It is possible to reduce state sizes by using compaction techniques. Also, it is often unnecessary to explore all execution paths to verify a given correctness claim. These techniques and others led to the development of powerful model checkers that can be used to verify complex concurrent systems.

A model checker can be used in different ways. A verified model can be developed first and an implementation derived from it, or a verification model can be derived from an existing implementation—a process that can be partially automated. In this thesis two specific

techniques are compared.

1. The more traditional approach: to construct and verify a model before the implementation phase. Specifically, the model and implementation are both structured as transition systems. This strategy makes it possible to verify the control flow of the *implementation* directly as opposed to verifying the control flow of the *model*.

2. Automated model extraction, using a simple technique proposed by Holzmann [28]. A parser for a specific implementation language is used to extract the control flow and a lookup table is used to remove irrelevant detail by replacing fragments of implementation code by abstract equivalents.

Thesis outline

The model checker SPIN was selected for this project because it is powerful and well documented [20]. The principles on which SPIN is based are discussed in Chapter 2. In addition, three typical industrial applications of SPIN are discussed to outline different strategies and to point out problems to be expected.

In Chapter 3 the technique of constructing a verified model and then deriving an implementation from it is discussed. The technique was applied to develop verified versions of the Alternating Bit and TCP protocols. Subsequently, the relatively new technique of automated model extraction was applied to the same two protocols. A model extraction tool was developed to derive verification models from existing implementation code. The outcome of this experiment is discussed in Chapter 4.

Experience gained from the two experiments is discussed in Chapter 5. The two techniques are compared in terms of ease of use, effectiveness and limitations.

Chapter 2

Concepts and applications of model checking

Several techniques have been developed to detect and eliminate defects in concurrent systems. Protocol engineering is one area where significant progress has been made. For example, formal specification languages such as SDL, Estelle and Lotos help to eliminate defects due to ambiguity in protocol descriptions.

Although formal descriptions are valuable, verification is needed to detect defects such as deadlock, unspecified receptions and duplicate messages in protocols. To help eliminate such defects, various computer aided techniques have been developed. Model checking is a particularly successful example of a computer aided verification technique that can be used to detect subtle defects in protocol specifications.

Popular specification languages such as SDL, Estelle and Lotos were not designed to support model checking. Although model checkers have been developed for subsets of these languages, it is difficult to handle the infinite structures supported by these languages efficiently. Specialised languages were therefore designed and adapted to simplify model checking. For example, SPIN—the model checker used for this project—accepts specifications written in Promela.

A model checker can be used in different ways to detect defects in protocol specifications. First, a model of a protocol can be constructed and verified before any code is written. Alternatively, a model can be derived from an existing implementation. Yet another option is to automatically generate models during the implementation process. This approach makes it possible to detect coding defects and provide useful feedback to programmers during the

implementation phase.

This thesis describes and compares two different styles of using a model checker. SPIN was used for the experiment because it is one of the most powerful model checkers available. Moreover, the algorithms used as well as various applications of SPIN are described in detail in the literature [7, 20, 33, 39, 44, 45, 49, 55]. Apart from that, valuable practical know-how regarding the SPIN model checker is distributed via the regular SPIN workshops [26]. A similarly detailed comparison of different styles of using a model checker and the relative merits of different approaches could not be found in the literature. The following techniques were investigated:

1. Verified models were constructed manually and used to derive implementations.
2. Automatic generation of models from implementation code.

2.1 The model checker SPIN

The model checker SPIN has been developed by Holzmann and was originally designed as a tool to verify computer protocols [20, 23]. In addition, SPIN has been used to verify widely different systems such as a micro-kernel [11], a flood surge control system [49] and even a business model [32]. Industrial case studies such as [44, 49, 28] illustrate the value of a tool such as SPIN in practical situations.

To verify a system, a model must be constructed. A model is an abstract representation of the behaviour of a system and should not include irrelevant detail. When using SPIN, models are described in the modelling language Promela (PROcess MEta LAnguage). The language has a C-like syntax and supports control structures loosely based on Dijkstra's guarded command notation [10]. An online reference manual for Promela is available at [15].

The well-known Alternating Bit protocol [3] will be used for illustrative purposes and specifically to introduce Promela as a notation for modelling protocols. Figure 1 shows a simple state diagram that describes the protocol.

The Sender process starts in state 0 and moves to state 1 by sending a message with a control bit labelled 0. An acknowledgement is then expected from the Receiver process. If an acknowledgement is received correctly, the control bit is altered to 1 and the procedure is repeated. Error management involves retransmission of messages. The Receiver process operates in similar fashion but to simplify the presentation message corruption is not supported,

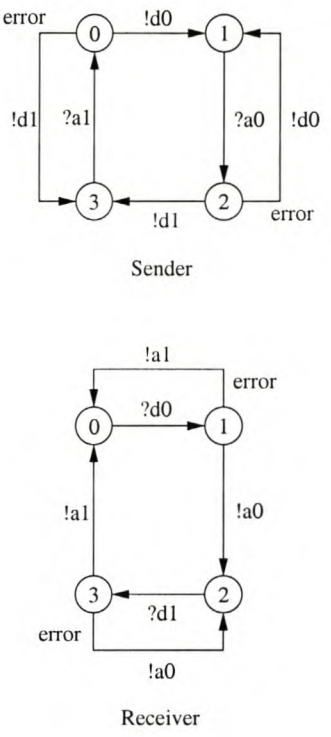


Figure 1: A state diagram for the Alternating Bit protocol. The symbols "!" and "?" are used to indicate the action of sending and receiving a message respectively.

only message loss.

Figure 2 shows a Promela model for the Alternating Bit protocol. There are three processes: **sender**, **receiver** and **init**. The **init** process is scheduled first and will usually create additional processes. The keyword **run** is used to create an instance of a process as shown in lines 48 and 49. Processes communicate by means of shared variables or message channels. In this example, the sender and receiver processes communicate via two channels *linkin* and *linkout*. Promela supports synchronous and asynchronous message passing. If the size of the buffer associated with a channel is zero, synchronous message passing is specified. This is indicated by "[**0**]" in the channel declarations.

The basic types **bit** (one bit), **byte** (one unsigned byte), **short** (a signed 16-bit value) and **int** (a signed 32-bit value) are supported in Promela. Structured types such as arrays and records are also supported. Control flow within a process is specified with the two constructs **do. . . od** and **if. . . fi**. The construct **if. . . fi** is used for selection. The selection structure shown in lines 8-15 has three command sequences, each preceded by a double colon (lines 9, 11 and 13). Only one of the command sequences will be executed at a time. A command sequence can

be selected only if its guard—the first command in the sequence—is executable. If more than one guard is executable, one of the command sequences is selected non-deterministically and if no guard is executable, the process blocks.

The functionality of the **do...od** construct is similar to that of the **if...fi** construct, except that the guards are re-evaluated indefinitely. Termination of a **do...od** construct must be specified explicitly with a **break** command.

SPIN can be used in one of two modes to analyse the model. First, a model can be simulated. The simulator interprets Promela commands on the fly and displays communication between processes in the form of a message sequence chart. Figure 3 shows a message sequence chart for the Promela code in Figure 2. The three vertical lines represent (from the left) execution sequences of the **init**, **sender** and **receiver** processes. A labelled arrow indicates a message transmission. The label indicates the channel number and the value(s) of the data element(s) of a message. The two values in the rectangular boxes are line numbers in the Promela code indicating the corresponding receive (top value) and send (bottom value) commands.

Simulation mode is used as a quick check to determine whether a model behaves as intended. The purpose is to eliminate major errors and in this respect the SPIN simulator is a valuable design tool.

Verification mode is used to determine whether a model satisfies a given correctness claim. If the correctness claim is violated, a counter example—an execution sequence leading to an error state—is generated. A counter example is normally displayed as a message sequence chart. An important and simple correctness claim to verify is deadlock freedom. SPIN automatically checks for deadlock (invalid end states), even if no explicit correctness claim is specified. Another simple correctness claim to verify is an assertion violation. The Promela **assert(boolean_condition)** command is always executable. If the boolean condition of an assert command does not hold, an error is reported during the verification process.

The temporal logic LTL is used in SPIN to describe more advanced properties that should hold along every execution path of a model. The logic extends propositional logic by adding a number of temporal operators. These include the unary operators $\Diamond$ (Finally) and $\Box$ (Globally) and the binary operator **U** (Until). A useful property that should hold for the Alternating Bit protocol can be stated as follows: If a message is sent, it should eventually be acknowledged. This property is described by the formula

$$p \rightarrow \Diamond q$$

where p and q are defined as **Data == 0** and **Data == 1** respectively. The variable **Data**

```
1  byte Data;

2  proctype sender(chan linkin; chan linkout) {
3      bit control, status;

4      Data = 0;
5      status = 0;
6      do
7      :: 1 →
8          if
9          :: linkout ! status, Data →
10             skip
11         :: linkout ! 1 - status, Data →
12             skip /* Corrupt the message */
13         :: 1 →
14             skip /* Lose the message */
15         fi;
16         linkin ? control;
17         if
18         :: control == status →
19             status = 1 - status;
20             Data++
21         :: else →
22             skip
23         fi
24     od
25 }

26 proctype receiver(chan linkout; chan linkin) {
27     bit control, status;
28     byte data;

29     status = 0;
30     do
31     :: linkin ? control, data →
32         if
33         :: control == status →
34             linkout ! control;
35             status = 1 - status;
36             printf("Data : %d", data)
37         :: else → /* Message was corrupted */
38             linkout ! 1 - status
39         fi
40     :: timeout → /* Resent previous acknowledge */
41         linkout ! 1 - status
42     od
43 }

44 init {
45     chan linkin = [0] of {bit};
46     chan linkout = [0] of {bit,byte};
47
48     run sender(linkin, linkout);
49     run receiver(linkin, linkout)
50 }
```

Figure 2: A Promela model for the Alternating Bit protocol.

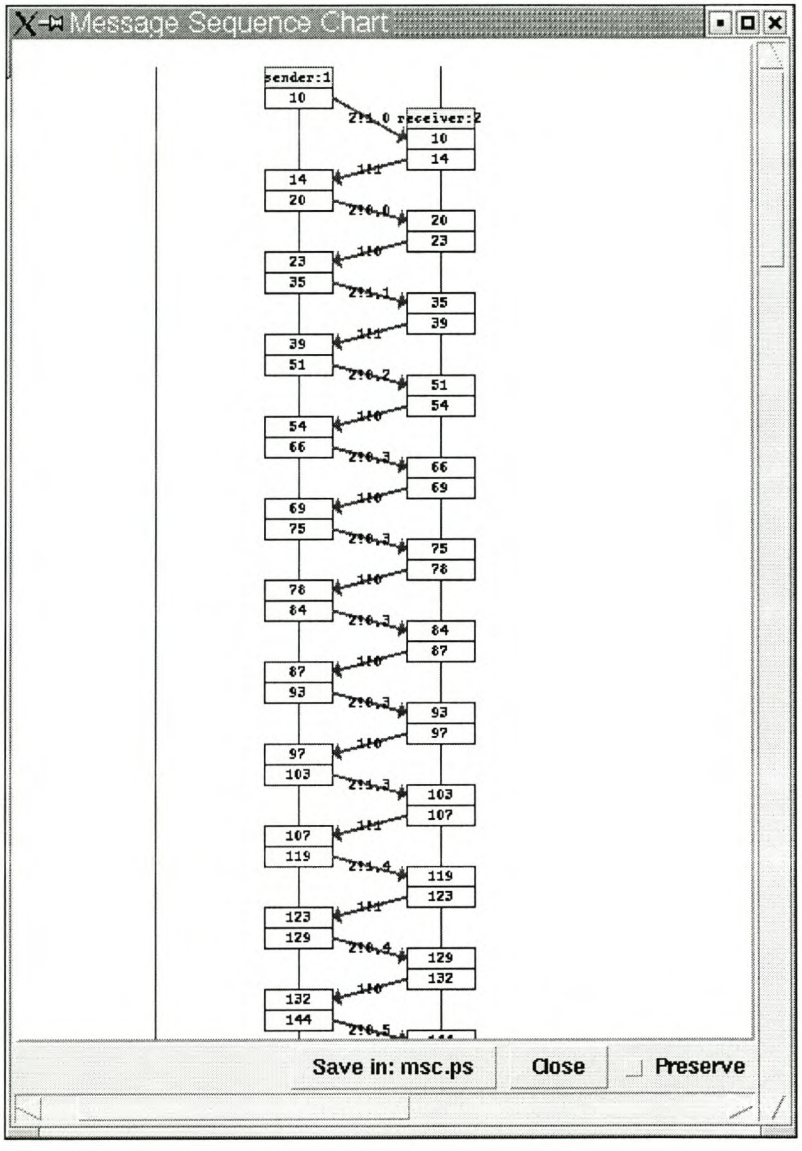


Figure 3: A message sequence chart generated by SPIN's simulator for the Alternating Bit protocol (Figure 2).

```
1  proctype sender(chan linkin; chan linkout) {
2      bit control, status;
3      byte lose, corrupt;
4      Data = 0;
5      status = 0;
6      do
7      :: 1 →
8          if
9          :: linkout ! status, Data →
10             corrupt−−;
11             lose−−
12         :: corrupt < 2 → /* Corrupt the message */
13             linkout ! 1 - status, Data;
14             corrupt++
15         :: lose < 2 → /* Loose the message */
16             lose++
17         fi;
18         linkin ? control;
```

Figure 4: The sender process of the Promela model shown in Figure 2 changed to eliminate the strong fairness problem.

will only be incremented if a message is successfully acknowledged after transmission.

SPIN can be used to show that this property is violated for the given model. The counterexample produced is an example of an unfair execution sequence. It shows that it is always possible to select the two guards in lines 11 and 13 of the model as shown in Figure 2. This is considered unfair because the guard in line 9 could also be selected when the other two guards are selectable. One way to eliminate the problem is to change the **sender** process as shown in Figure 4. The two variables *lose* and *corrupt* are used to ensure that the verifier does not always select the guards in line 12 and 15.

In [14, Chapter 0] Francez defined several different versions of fairness. Weak fairness requires that any transition that is permanently enabled must be executed eventually. On the other hand, strong fairness requires that any transition that is enabled infinitely often must be executed infinitely often. SPIN supports weak fairness. However, selecting this option when verifying the model described above will not solve the problem. The reason is that none of the transitions (lines 9, 12 or 15) is permanently enabled because selection of any one immediately disables the others.

2.2 Internal mechanisms of SPIN

SPIN is an automaton-based model checker. Both the model (description of the behaviour of the system) and the property to be checked are represented as finite automata on infinite

execution sequences. These are known as Büchi automata. The synchronous product of these two automata describes all legal execution sequences. Correctness properties in SPIN are described as temporal logic formulas and these are converted into Büchi automata. However, instead of representing a correctness claim directly as a Büchi automaton, it is negated first. This means that the corresponding Büchi automaton now describes an error pattern. The synchronous product of this automaton (corresponding to the negated correctness claim) and the automaton that represents the system behaviour now describes all execution sequences that violate the given property. Introductions to the theory of automata-based model checking can be found in either Bérard [4], Clarke [13] or Peled [42].

The central problem of model checking is that some models simply generate too many states. This is known as the state explosion problem. In practice, the most serious limitation is the amount of main memory that is available because this determines how many previously visited states can be stored.

Three of the most successful techniques used in SPIN to combat the state space explosion problem are:

1. **Bit-state hashing:** In 1988 Holzmann introduced the technique of **bit-state hashing** [19]. The basic version of this technique (single-bit hashing) uses all available memory to store a large bit-array. Every generated state is hashed to an index in the table and the bit at that position is set to one. If the bit is already set, it is concluded that the state has already been visited. The disadvantage of this technique is that hash collisions can occur. This means that errors may be missed because search paths are terminated prematurely. Using several independent hash functions may seem like an obvious solution to this problem. The idea is to terminate the search only if all bits, corresponding to every hash function, are set. This strategy was analysed by Wolper in [54]. Holzmann concluded that the best approach is to use two independent hash functions and two bits per state [29].

2. **Partial order reduction:**

   This technique reduces the amount of work during model checking by eliminating paths that lead to the same states simply by scheduling processes in a different order. Therefore, instead of constructing the full state graph, a reduced state graph is constructed. Some of the first work on partial order reduction was done in the late eighties and early nineties by Valmari, Godefroid and Wolper [17, 50]. Which paths are preserved during the reduction process depend on the property being checked.

Partial order reduction was added to SPIN in 1994. Holzmann and Peled introduced a **static reduction** algorithm which performs the computations needed to apply partial order reduction before the search, instead of during the search. This approach was shown to have significant advantages over dynamic reduction techniques [27].

3. **State compaction:** As the speed of processors increased, it became possible to employ algorithms that use a little more processor time to save memory. During 1995, a **state compression** technique called collapse mode, was added to SPIN. The technique exploits the observation that the local variables of each process will be assigned only a relatively small number of distinct values which are called **local states**. A much larger number of states is produced by the numerous ways in which these local states are combined to obtain **global states**. With state compaction, each local state vector component is stored separately in a local hash table. The compressed global state vector consists of the index numbers of the local state vector components in the local hash tables. The global state vector is then again hashed to a global hash table.

   The collapse mode in SPIN requires the user to specify an upper bound on the index number range for the local state vector components. The required guess of the range prohibited this method from being promoted to the default compression mode in SPIN. A revised method, called Recursive Indexing, was later added that stored both the component index-number and the number of bytes used per component index in the global state vector. With this technique it is not required to specify an upper bound on the number of component indexes. Holzmann states that the recursive indexing method is a relatively simple and robust method that can give good reductions, in return for a doubling or tripling of the run-time costs [21].

2.3 Applications of SPIN

This section is a discussion of how SPIN was used to verify two different protocols, illustrating the approach of applying model checking manually. The two examples were chosen for the following reasons:

1. Both examples are complex, well documented protocols.
2. The first example illustrates how model checking is typically used to verify a specific property of a protocol.

3. The second example illustrates how model checking was used to verify the design of a life-critical system in actual use.

A third example, described in Section 2.3.1, illustrates a different approach: Derivation and verification of a model from implementation code. This approach is called automated model extraction.

High Performance Error Control Protocol

HTPNET is a protocol designed to exploit higher communication bandwidth while maintaining low error rates [7]. The lower error rates led to a design where synchronisation packets are periodically sent between data packets—as opposed to a design where each data packet contains synchronisation information. The protocol was verified but not implemented. For a rough estimation of the protocol's complexity, it is compared to the rather complex Transmission Control Protocol (TCP) [47]. HTPNET and TCP have similar connection management techniques, the three-way handshake and graceful termination. The main difference between the two protocols is the error detection strategy.

The authors described only one model consisting of four processes: two hosts, a transmitter and receiver, and two processes to handle error correction. Two message channels were used to connect the transmitter and the receiver, simulating a full-duplex network. The model focused on the error correcting protocol. The connection management techniques were left out to produce a tractable model.

The model was verified using assertion violations and end labels. An iterative method was followed where the model was first verified under error-free conditions and then against a network losing packets at random. Under error-free conditions the model was verified using an exhaustive analysis. With the introduction of a lossy connection, the state space was increased dramatically. The increase was such that an exhaustive search could not be performed with the available memory. To reduce memory usage, SPIN's super-trace feature was used. This uncovered an error after inspecting more than forty million states. With the super-trace option only 5MB of memory was used to detect the error. According to the authors an exhaustive search would have required more than 6336MB of memory.

Summary

The authors followed a verification approach where a manually constructed model was verified first before the implementation phase began. Unfortunately the protocol was only verified and

not implemented. It is therefore not clear how the correctness of an implementation would be influenced by the verified model. It is however possible to argue about the relationship between the model and a possible implementation if the structure of the model is studied.

The Promela code shown below is an excerpt from the model.

```
1   do
2   :: errcnt_to_host[1]?white, 0
3   :: (r_state == 0 && errcnt_to_host[1]?[red]) ->
4        errcnt_to_host[1]?red,0 ->
5        r_state = 1
6   :: (r_state == 0 && errcnt_to_host[1]?[blue]) ->
7        assert(0)
8   :: (r_state == 1 && errcnt_to_host[1]?[blue]) ->
9        errcnt_to_host[1]?blue,0;
10       break
11  :: (r_state == 1 && errcnt_to_host[1]?[red]) ->
12       assert(0)
13  od;
14 end:
15  do
16  :: errcnt_to_host[1]?white,0
17  :: errcnt_to_host[1]?red,0 -> assert(0)
18  :: errcnt_to_host[1]?blue,0 -> assert(0)
19  od
```

This code is central to the receiver host process. The sender process sends three different colour coded messages in the following order: one red message, one blue message and randomly inserted white messages. The receiver process accepts the sent messages and aborts the process if a message is received that is out of order (lines 7, 12, 17 and 18). Popular implementation languages used in industry, such as C, do not support or resemble the code shown in this example. It is difficult to imagine how a designer will use the model as a guide for a detailed design. The logical consistency of the protocol was verified but the model does not serve as a structural guide for the implementation. It is therefore still possible for a designer to introduce subtle coding errors.

The authors verified the model for deadlock, livelock and assertion violations. No mention is made of any LTL formulas that were checked. This gives the impression that the protocol

was verified insufficiently. However, Holzmann advises that when a model is built, the first step should be to decide which correctness properties are relevant. The model should then be constructed to capture the behaviour of the system necessary to prove the correctness properties chosen in the first step [24]. The completeness of the verification phase therefore depends on whether the identified properties were verified.

In this example the relevant property identified by the authors was the protocol's ability to deliver messages in the correct order over a lossy connection. They argued that it suffices to use only three distinctly numbered messages to efficiently verify the mentioned property. The model was specifically constructed to verify this one particular property. The property being verified is captured in the logic of the model and it is therefore not required to specify LTL formulas.

Finally, subtle errors were detected in this protocol. One particular error was detected only after visiting more than forty million states using SPIN's super-trace technique. This is because the error only emerges after a consecutive loss of state synchronisation packets under conditions where no data packets are sent. It would be extremely difficult, and most unlikely, to reconstruct this particular error trace manually and would probably not have been detected if conventional testing techniques were used.

A Storm Surge Barrier Control System

SPIN was used in the verification efforts of a concurrent system called BOS (Dutch: Beslis en Ondersteunend Systeem). The BOS system is responsible for the correct functioning of a movable storm surge barrier built near the Hook of Holland in the Nieuwe Waterweg. The system consists of several subsystems communicating with one another using different protocols. The operational system consists of 200 kLOC (thousand Lines Of Code) and an additional 250 kLOC for simulators, test systems and supporting software [16, 37, 38, 49].

A combination of Promela and Z was used to specify the system. The control flow was expressed in Promela and data manipulation in Z. Models of each subsystem were then constructed using the Promela/Z specifications: The control flow was left as is and the Z specified data manipulation fragments were translated to Promela code. Most of the validation models consisted of three or more processes: one process to model the interface, another process to model the environment and one or more control processes. The models built were kept as close as possible to the specifications, but some features had to be simplified to produce tractable models. For example, the BOS process (the process responsible for the decision

making) communicates with the BESW process (the process that performs the task issued) by means of polling. When a command is issued, polling requests are used to wait for the completion of the issued command. This mechanism was replaced by waiting for a variable to reach a certain value.

An incremental approach was followed: Initially a design and an environment were verified with no error behaviour. Some error behaviour was then introduced and after each successful verification run more were added. The models were only checked against a restricted set of properties which included the absence of live-lock and deadlock.

Summary

The general model checking approach in this example is similar to that of the previous example: Both the protocols were verified before the implementation phase began. However, this example has two noticeable differences compared to the previous example. The first is that the system was implemented and not just verified, and the second has to do with the rationale behind the structure of the models.

The fact that the system was implemented enables one to determine how the design phase was influenced—if influenced at all—by the models. To determine the amount of influence, the structure of the models should first be examined. The structural design of a model is determined by one of two things: A model is either constructed to capture the control flow of a system necessary to verify identified properties, or a model is constructed to find the correct structure for a design.

If a model is built for the purpose to verify an identified property, only code that is needed to verify the property is added to the model. This means that unnecessary code is omitted from the model and the probability of building a tractable model is increased. However, in producing such models it is often required to use advanced abstraction techniques. The use of abstraction techniques lead to models that are difficult to read and understand—especially if a model was constructed by someone else. As with the previous example, the authors were not concerned with the structural design of the protocol, only in the correctness of the protocol's procedure rules. This led to a model design that had little resemblance to the specification.

A model constructed to find the correct structure for a design limits the chances of a designer adding errors to the system through a faulty design. Although this approach seems ideal, there are two potential problems: First, it may be difficult to find the correct structure and still produce a tractable model, and second, the model may not be verified sufficiently. It is often impossible to construct a tractable model that captures all the behaviour of a large

system. Such a system should be divided into smaller parts. How this is done and how models are produced that sufficiently capture the behaviour of the system, requires experience.

The second problem is inherent to testing and verification. In [40] Myers argues as follows:

> ... testing is a destructive process. In other words, it is extremely difficult, after a programmer has been constructive while designing and coding a program, to suddenly, overnight, change his or her perspective and attempt to form a completely destructive frame of mind toward the program.

It is therefore advisable to appoint an independent test team to perform the testing phase in a software project. This is also true when applying model checking (with the goal to find the correct structure of a design). Building a model is a constructive art, verifying it is destructive.

In this example the verification team tried to structure the models as closely as possible to the specifications. This implies that they were not only concerned with the logical correctness of the protocol's procedure rules but also with the structure of the design.

According to the authors, the models were checked against a restricted set of generic properties such as progress properties and the absence of deadlock and live-lock. As with the previous example, the use of model checking in this project uncovered many errors, ambiguities, inconsistencies and examples of incompleteness that would have been difficult to find with conventional testing methods. The authors state that most errors were found during manual analysis of the specifications. Some errors were found during simulation. Even so, a few errors remained undetected and were only revealed during model checking. This shows that although a model checker cannot guarantee the correctness of a system, it is a particularly effective tool to uncover those errors that are rarely found by analysis of the specifications and conventional testing methods.

2.3.1 Automated Model Extraction

The examples described in the previous section illustrated how model checking is applied manually. This approach has several disadvantages and Holzmann states three important negative issues when following this approach in [28]:

1. Constructing a model manually requires time, skill and experience. It is therefore difficult to perform this type of verification during the design and development phase,

when verification results are most beneficial. For this to be possible, the effort to construct a model must be considerably smaller than the development effort itself.

2. When verification results become available, chances are that they apply only to older versions of the code, which may no longer be of interest to the designers.

3. Manually constructed models can themselves be flawed, possibly hiding real errors and often introducing artificial errors. The latter can be detected and corrected, the former cannot.

Constructing a model to find the correct structure for a design, as described in the last example of the previous section, contains yet another potential problem: There exists no formal derivation method leading from a verified specification to an implementation. A coder following a complicated model as an implementation guideline may err in his/her derivation from the modelling language to the implementation language.

A current focus of research is the automatic derivation of models from source code. A direct approach is to develop a translation tool that can translate a given implementation language to a tractable model in some modelling language. AX (Automaton eXtractor) is an example of an extraction tool [28, 25, 31]. This tool translates programs written in ANSI-standard C to Promela models. Other approaches on applying automated model extraction are discussed in Chapter 4.

AX consists of a scanner, parser, lookup table and a file containing a model template. Figure 5 is a schematic representation of how the different components are combined to produce a model. First, a model template is constructed manually. This template contains an outline for the required data declarations and an outline of the required process declarations. A lookup table is then constructed manually. This table contains entries that are mappings from C statements to statements in Promela. Each entry defines the abstract Promela code associated with the corresponding C statement(s). Finally, the C source code is given as input to the scanner. The scanner and parser will extract the control flow from the source code. The rest of the detail is added to the model using the lookup table and model template.

The PathStar Access Server

The PathStar Access Server is a system that supports call processing, telephone switching and data connections in an IP network. The system consists of several software modules and one of these modules, the call processing module, was verified with the AX tool. The call processing

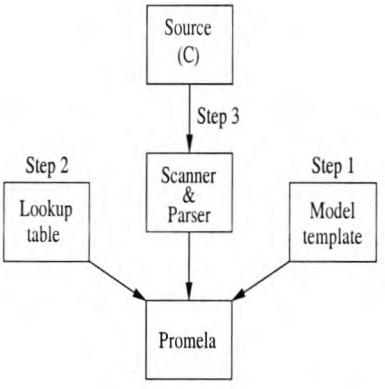


Figure 5: Schematic representation of the model extraction process.

module was developed over a period of nine months and the complete system was developed over several years. According to the authors this is the largest commercial product that has been verified using software model checking up to the time of the publication (1999) [28].

The authors started the extraction process after the first C implementation of the call processing module was available. A lookup table and model template were constructed and the first model was produced. The model was verified and errors found were reported to the implementation team. With each update of the source code a new model was produced ready for verification. Small changes that were made in the source code required little or no updating of the lookup table. With a few exceptions, each update required no more than five minutes. With major changes in the source code the update of the lookup table required only a few hours.

Each version produced until the release of the call processing code was verified using the extraction tool and every model extracted was checked against the same set of LTL properties. A database was constructed that contained over one hundred properties. Apart from the properties, the database also contained additional information such as the result of a verification run, the values of environment variables and other information necessary to perform a verification run for a specific property.

Summary

The automated model extraction approach followed in this example proved to be successful in finding errors fast and effectively during the implementation phase. The success can mainly be attributed to the structure of the design and the format of the implementation code. The design of the system was structured such that the central code related to call processing—the code verified—was concentrated in a single high-level procedure. This procedure could

be studied and verified separately from the remainder of the code. This means that it was possible to construct a meaningful, tractable model without the need to include behaviour from the rest of the system that would have increased the number of reachable states.

The implementation code was structured as a state machine. The state machine executes transitions which consist of a source state, guard condition, actions and destination state. This type of code structure separates the control flow from data manipulation: The control flow is specified by ⟨ *source state, guard condition, destination state* ⟩ triples and data manipulation is performed in the actions. The lookup table contains entries that specify mappings from source statements—that manipulate data—to statements in the modelling language. The construction of the lookup table is therefore simplified because all the data manipulation code is contained in the actions. In Chapter 3 the approach of separating control flow from data manipulation is discussed in detail.

Three factors contributed to minimise the time spent verifying the validity of each property:

1. The construction of a model was partially automatic. The only manual tasks performed were to construct a lookup table and a model template.

2. SPIN always generates a model-specific verifier in C for a complete specification which consists of a part for LTL and a part for Promela. To save time in compiling a complete verifier for each property in the database, the authors added an option to SPIN to separately generate and compile the LTL code from the model code. The generated and compiled LTL code is therefore reused which, according to the authors, reduced the compilation time for all properties from minutes to seconds.

3. An iterative search refinement procedure was followed to find errors as quickly as possible. Initially the bit-state search option was used with a small hash table. If no errors were found, the size of the hash table was doubled. The authors claim that errors typically showed up in the first few iterations of a search. Properties that were likely to be satisfied were the only ones that had gone through the entire iteration process, from a fast approximation to an exhaustive search.

The authors report that during the development of the code several errors were discovered by the implementation team that they could not reproduce. These errors were uncovered with SPIN by specifying the pattern of events that had proceeded each observed error. With the help of the error traces that SPIN generates, each error could be diagnosed and repaired. In this example model checking thus did not only serve as an verification method to verify abstract properties, but also as means to diagnose errors.

Only the positive results were discussed in this section, but there are several problems that can arise when this particular method is applied. For instance, the code in this example was structured to facilitate the setup of the lookup table because data manipulation code was contained in units called actions. If the source code had been written such that the data manipulation and control flow code were not easily separable, the setup of the lookup table would have become a difficult and time-consuming task. In Chapter 4 these issues are discussed in more detail.

Protocols in embedded systems

In the last two sections it was shown how SPIN was used to verify protocols in practice. In the rest of this thesis the scope is narrowed to focus specifically on protocols in embedded systems.

Embedded systems are often produced for the mass market and have little memory and relatively slow CPUs to keep costs down. This means that the software on such systems should be optimised for speed and memory usage. Also, finding errors in the software before releasing a large number of systems can lead to significant cost reduction.

A well known method for implementing protocols is to structure the design of a protocol as a transition system. In the next chapter we will illustrate how this approach can be combined with manually applied model checking to implement reliable protocols in embedded systems.

Chapter 3

A transition system approach

In this chapter an approach to apply model checking manually is discussed. The technique followed restricts the design of a system to a formal structure. In [24] Holzmann suggests the following guidelines when applying the model checking approach where a design is first verified and then implemented:

1. First identify which correctness properties are relevant, and require formal proof.
2. Second, study the essence of the solution that is used in the application to secure the behaviour of interest (that is to say, the behaviour that determines correctness with respect to the properties selected in the first step).
3. Finally, construct an executable abstraction for the application that has enough expressive power to capture the essence of the solution and no more.

In general it is difficult to apply step 3 during verification of a realistic size system because the amount of detail could lead to an intractable model. Abstraction techniques are therefore required to reduce the size of the state space. Applying abstraction techniques often lead to models that have little resemblance to the actual structure of the implementation. It may therefore be meaningless to use the model as a structural guideline for coding.

To relate models and implementations more closely, we have suggested to structure both a specification and its implementation as a transition system [34]. The restriction on the structure allows for the separation of control flow and data manipulation as will be discussed in the next section.

3.1 Implementing Protocols as Transition Systems

In [1] Arnold defines a transition system as a quadruple $\mathcal{A} = \langle S, T, \alpha, \beta \rangle$ where:

- S is a finite or infinite set of states,
- T is a finite or infinite set of transitions,
- α and β are two mappings from T to S which take each transition t in T to the two states $\alpha(t)$ and $\beta(t)$, respectively the source and the target of the transition t.

A transition t with source s and target s' is written $t : s \longrightarrow s'$. Several transitions can have the same source and target, i.e., the product mapping $\langle \alpha, \beta \rangle : T \longrightarrow S \times S$ is not necessarily injective. A transition system is finite if S and T are finite.

Execution of a given transition system starts in a well-defined initial state from which different states are reachable by executing transitions. A finite set of possible events is associated with each transition and the occurrence of one or more of these events is said to enable the corresponding transition. Only enabled transitions are executable. The graphical representation of a transition system, also called a reachability graph, consists of nodes which represent unique system states and edges which represent transitions between states. A unique action is executed in response to each event. An action is a small, independent code fragment that implements operations that may modify the system state.

Since a transition system is such a low-level formalism, it is suitable for direct execution through interpretation. The behaviour of a transition system is uniquely defined by its current state and its transition relation. A table driven approach is used to store the transition relation. This approach allows for a compact representation of the transition relation without adding too much overhead for interpretation.

The implementation of the table driven approach consists mainly of four parts:

1. Two global variables, **State** and **Event**.
2. An interpreter.
3. A transition table.
4. Actions.

The variable **State** is used to keep track of the current system state and variable **Event** is used to keep track of the event that has happened most recently. Typically, the events of a protocol can be classified into two categories: external events (such as interrupts) and internal events (such as checksum errors). An action can be seen as a procedure that has no formal parameters. Most importantly, it executes independently and is atomic. The purpose of an action is to modify system data and to assign a new value to the **Event** variable. The new value indicates which event occurred during the execution of the action. A transition table entry contains an event number, a pointer to the associated action, a new state number and a pointer to a chain of alternative events (zero if none) that could happen in the state.

The interpreter, which combines the four parts discussed above, works as follows: the current state number is used as an index into the table to retrieve the appropriate table entry. If the first element in the chain of events corresponds to the event that has happened, the appropriate action is executed immediately. Otherwise, the chain is followed until the appropriate action is located or the end of the chain is reached (which signifies an error).

Structuring a protocol as a transition system is not an unfamiliar practice and specifications of protocols are often presented in the form of a transition system. The transition system structure is also suitable for verification: a model checker explores the reachability graph of a given transition system to detect subtle errors in its behaviour. With this approach the system is verified at implementation level because the model and the implementation have the same structure. Finally, testing is simplified because it is restricted only to actions. An action is a small code fragment that does not depend on external events.

3.2 The Alternating Bit Protocol

Assembly language is generally viewed as an impractical implementation language for large systems mainly because the code becomes unmanageable. However, such low-level code optimises execution speed and memory usage. Such optimisations are typically required on small embedded systems with little memory and relatively slow CPUs.

We have implemented the Alternating Bit protocol in assembly language to illustrate how the proposed technique can be used to implement small, manageable and optimised embedded systems. Also, the simplicity of the protocol is ideal to illustrate the principles of the approach without introducing complex procedure rules. Other issues illustrated include the estimation of memory requirements, the overhead imposed by interpretation, and to what extent verification at the implementation level is possible.

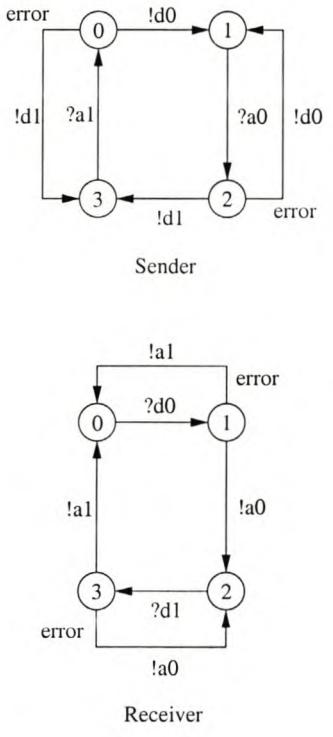


Figure 6: Transition system describing a simple version of the Alternating Bit protocol.

The first step in following the proposed technique is to identify the states, events and actions of the protocol. This step is simplified if the description of a protocol is expressed as a reachability graph, where nodes represent states and edges transitions. Figure 6 shows a reachability graph for the Alternating Bit protocol that was defined in Section 2.1. Each edge is labelled with an action that is associated with the transition. In the sender the label **!d0** represents sending data message 0 and **?a0** represents receiving acknowledgement 0. Each transition must have an enabling event. For example, five events are identified for the sender: data message 0 was sent (event 1), acknowledge message 0 was received (event 2), data message 1 was sent (event 3), acknowledge message 1 was received (event 0) and an error occurred (event 4).

Using the reachability graph in Figure 6 as guideline, the entries of the transition table for both the receiver and the sender can be determined. Figure 7 shows a schematic representation of the transition table for the Receiver process. The labels e_i, s_i and a_i represent the event number, the next state and a pointer to the action associated with the event respectively. The fourth label of an entry, which is either a **-1** (for none) or an arrow, indicates the next event associated with the current state.

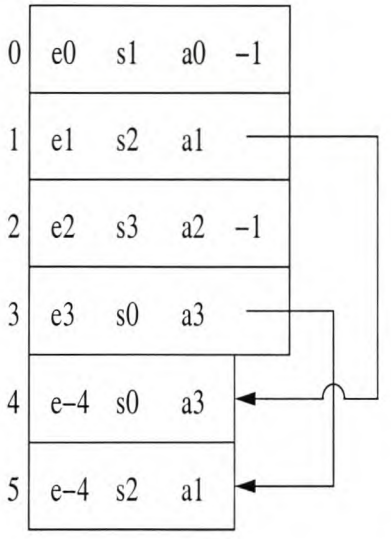


Figure 7: A schematic representation of the transition table for the Receiver process of the Alternating Bit protocol.

To represent the table as compactly as possible, all the entries—except the first one—that are associated with a state, are stored consecutively in memory. The last entry of a list associated with a state is indicated by a negative event number. For example, in Figure 7 entry 4 in the table is the second entry associated with state 1 and it is also the last entry in the list (**e-4** indicates a negative event number).

Next, a Promela model must be constructed. The model for this protocol consists of a process to represent the sender, a process to represent the receiver and a third to represent the communication link. The transition table and the interpreter are combined in the model using a single Promela **do . . . od** construct. The translation from the schematic representation of the transition table in Figure 7 to the Promela code shown in Figure 8 is a trivial exercise. Actions are modelled using Promela macro definitions. The Promela code for the action that corresponds to the **?d0** label in the receiver of Figure 6 is shown in Figure 9.

The communication link is modelled as a separate process. Generally this is not a good approach: to reduce the size of the state space the behaviour of a communication link should be modelled in the processes that communicate over the link. However, the goal with the transition system approach is to relate the model and the implementation as closely as possible. The complete model is presented in Appendix A. This model was verified against correctness properties such as freedom from deadlock and correct behaviour if messages are lost or corrupted.

The final step is to translate the verified model to the implementation language. In this

```
proctype Receiver(chan link_in, link_out) {
    do
    :: (state == 0) && (event == 0) →
        state = 1; RAction0()
    :: (state == 1) && (event == 1) →
        state = 2; RAction1()
    :: (state == 2) && (event == 2) →
        state = 3; RAction2()
    :: (state == 3) && (event == 3) →
        state = 0; RAction3()
    :: (state == 1) && (event == 4) →
        state = 0; RAction3()
    :: (state == 3) && (event == 4) →
        state = 2; RAction1()
    od
}
```

Figure 8: The Receiver process in the Promela model of the Alternating Bit protocol.

```
/* Receive data packet 0 */
#define RAction0() \
    if \
    :: Serial_In ? data → \
        if \
        :: data == 0 → \
            event = 1 /* indicates that data packet 0 was received * \
        :: else → \
            event = 4 /* indicates error event */ \
        fi \
    :: timeout → \
        event = 4 /* indicates error event */ \
    fi
```

Figure 9: Promela code for the actions of the receiver process that corresponds to the **?d0** label of the receiver in Figure 6.

```
1  Action0:
2  ;(* This action sends data packet d0 *)
   ...
   ...
3  jmp Interpreter                ;Action returns

4  Action1:
5  ;(* This action waits for the acknowledgement a0 *)
   ...
   ...
6  jmp Interpreter                ;Action returns
       :
7  Interpreter:
8  lea edi, [TransitionTable+esi*8]Find the first record in the list
   ...
9  mov word si, [di + 2]          ;Enters next state
   ...
10 jmp [di + 4]                   ;Jump to the action
   ...
11 jmp $                          ;Endless loop if error occurs
       :
12 ;(*————————————————————*)State 0
13 TransitionTable  dw 0          ;Event
14                  dw 1          ;Next state
15                  dw Action0    ;Action to execute
16                  dw addr1      ;Next Field
17 ;(*————————————————————*)State 1
   ...
   ...
18 ;(*————————————————————*)
19 addr1            dw -4         ;Event
20                  dw 3          ;Next state
21                  dw Action2    ;Action
```

Figure 10: Assembler code for the sender process of the Alternating Bit protocol.

example it was possible to translate the code manually, but the task should be automated for more complex systems to eliminate errors and to save time. The control logic of the interpreter—as described in the previous section—is simple and its implementation required only 26 lines of assembly code (Lines 7 to 11 in Figure 10 shows the outline of the interpreter implemented in assembly language). The transition table was also easily translated. Lines 12 to 21 in Figure 10 show the outline of the transition table for the sender process implemented in assembly language.

The actions for data manipulation must also be coded manually. The danger is that errors may slip in, but the advantage is that the code can be optimised for efficiency. Also, the task is simplified by using the Promela code of each action as guideline during the implementation. Furthermore, it is possible to test each action separately. The functionality of each action is specified by providing a precondition-postcondition pair and standard techniques like equivalence class analysis and boundary value analysis are useful to derive effective test cases for each action [35].

The transition system structure simplifies testing of each code fragment. The system is simply initialised to enable the particular action to be tested. It is then useful to display the result of each action and check it against the specification. For example, the action that performs sending a packet in the sender process has the following precondition-postcondition pair:

- Precondition: The **Event** variable must be equal to *AckReceived*.
- Postcondition: The send buffer is filled with data, the control bit and checksum are added and the event variable is set to indicate that a packet was sent.

Debugging code that displays the contents of the send buffer and the values of the control bit, the checksum field and the event variable was added to the action. The action was then executed by assigning the appropriate values to the **State** and **Event** variables. The final step was to examine the output displayed. A reactive system implemented in the traditional way is significantly more complex to test since it involves control flow analysis, coverage analysis and unpredictable external events.

The experience gained with this experiment showed that the transition system approach consists of several stages. The actual size of each action can be determined only after the translation process from the model to the implementation language has been completed. In this example, the implementation sizes of the actions used to send and receive messages were identified as being too large: Both these actions include computing a checksum. It is less error-prone to test the action of computing a checksum separately. Each of the mentioned actions were divided into two separate actions: an action for sending/receiving data and an action for computing the checksum. The revised transition graph for the receiver is shown in Figure 11. Updating the model, verifying it and translating the code required little time and effort.

3.2.1 Discussion

Verifying and implementing the Alternating Bit protocol structured as a transition system revealed three key points:

1. Memory estimation

The formality of the transition system design allows for the memory requirements to be estimated rather accurately even before an implementation is attempted. Memory is needed

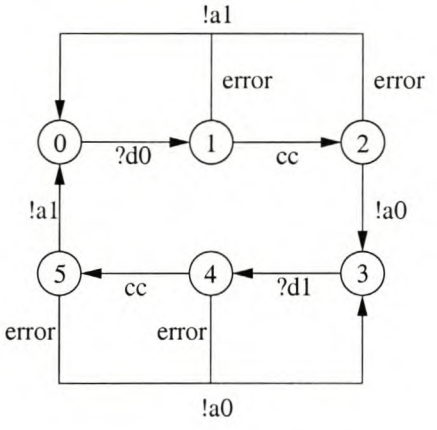


Figure 11: A revised transition graph for the receiver process of the Alternating Bit protocol.

for the following entities:

- The code for the interpreter. The control logic of the interpreter stays the same for any system implemented using this approach. The amount of memory required for the interpreter depends on the implementation language used and the architecture of the target machine. In this example the interpreter required only 57 bytes implemented in assembly language on the IBM PC.

- The transition table. The transition table can be seen as a table of records. There are two types of records: the first, consisting of four fields, is used to store the first event–action pair associated with the current state. It also contains a field for the next state and a pointer to possible additional entries of a second record type. The second record type contains information about other events that may also occur in the current state. This record type contains only fields for the event, the next state and a pointer to the associated action. The pointer field to a next record is not required because the rest of the records associated with the current state are stored sequentially.

  For the implementation of the Alternating Bit protocol each field was implemented using 16 bits. This should be enough to represent the number of states and events for rather complex protocols. Exactly how many entries are required in the transition table can be determined from the specification of the protocol. One entry is needed for each transition.

- Code to implement the actions. Each action is a small code fragment and exactly how

much memory will be required is determined by the complexity of each action. The number of actions are known from the reachability graph. Multiplying this by the average memory requirements per action provide a reasonable estimate for planning purposes. A complex protocol can reach hundreds of different sates and, depending on the environment, may involve hundreds of different actions. Therefore, memory requirements are mostly determined by the number of transitions and actions.

The final argument presented in the previous section suggested that large actions be divided into smaller fragments to facilitate testing. This fragmentation does influence the memory estimation. For each new action an entry must be added to the transition table. Still, for this example, each entry does not occupy more that 8 bytes and a first approximation will not differ much from the final version.

- Global memory areas for data storage. Memory areas are needed to store packets, various counters, etc. The amount of memory needed depends on the particular protocol.
- Memory for the run-time system. It is possible to implement device drivers in the same fashion as protocols, interrupts simply representing events and the driver itself a number of states and actions. This approach was not followed in the experiment, however. The drivers needed to support the protocol were implemented in the traditional way as part of the supporting environment.

2. Reliable code

The transition system approach improves the reliability of the code for several reasons:

- Computer-aided techniques, such as model checking, can be used to check specifications to detect subtle errors. In fact, model checking can be used to verify the flow of control at the code level because a table driven implementation technique is used.
- It is feasible to test the code that implements the actions thoroughly, as shown in Section 3.2. In comparison, if traditional implementation techniques are used, it is difficult to reach the same level of test coverage even when using sophisticated test tools. With the proposed technique control flow analysis is unnecessary because that is done automatically during verification.
- Specifications in the form of verified models can be translated automatically into implementations where the only manual coding required is to implement the actions that manipulate data. Because this technique is so simple, manual translation was used

to develop the implementation for the Alternating Bit protocol, but automating this process involves little more than parsing the verified model and generating a transition table to drive the interpreter. The input to such a translation tool would be a verified model written in Promela.

3. Maintainable code

Finally, the transition system approach allows for easily maintainable code. Descriptions of protocols as transition systems are easy to understand. The verified model documents the control structure of the system and if anything must be changed, it is simple (and advisable) to recheck the model before the transition table is modified. If the details of an action must be changed, it is also simple to rerun the tests for that action. This approach ensures that the current version of the implementation always correlates to the specification/design.

3.3 The TCP Protocol

To investigate how the transition system approach would scale for larger applications, it was decided to model the Transmission Control Protocol (TCP). The control logic and data manipulation of the protocol are reasonably complex and a typical implementation in C—such as the UNIX version—consists of more than 4000 lines of code [48, Chapter 24]. For this example, C was used as implementation language to simplify interfacing with the underlying Internet Protocol (IP) in a UNIX environment.

3.3.1 An overview of the TCP protocol

TCP is a connection-oriented, sliding window protocol. A TCP process—which is a TCP implementation being executed—progresses through three phases: connection establishment, data transfer and connection termination. Each of these phases can be studied independently and a short summary of each is presented:

1. Connection establishment phase: The TCP protocol uses sequence numbers for control flow. Each byte in the user data sent is associated with a unique sequence number. Sequence numbers are used to detect lost, old or duplicate packets and to govern the amount of data that each TCP process can manage during a connection. The sequence number range of user data starts from a computed initial number and two TCP processes

that want to communicate should know each other's initial sequence number before data is being transfered.

To exchange initial sequence numbers, TCP makes use of what is called the "three-way handshake". The "three-way handshake" symbolises the notion of three synchronisation packets that are sent between two TCP processes to establish a connection: The first packet sent to a receiver TCP process contains the initial sequence number of the sender TCP process. The second packet sent acknowledges the first packet sent and also carries the initial sequence number of the receiver TCP process. The third packet sent acknowledges the reception of the receiver's initial sequence number. Once the "three-way handshake" is completed a connection is established.

2. Data transfer phase: During this phase the bulk of the user data are transferred and each message sent should be acknowledged. The TCP protocol uses a form of flow control called the sliding window protocol. It allows for a sending TCP process to transmit multiple messages before it stops and waits for an acknowledgement. This leads to faster data transfer since the sender process does not have to stop and wait for an acknowledgement each time a message is sent.

   Messages that get lost in the network are retransmitted using a retransmission timeout. An acknowledgement time is associated with each message sent. If a message sent is not acknowledged before its retransmission timeout, the message gets retransmitted. Duplicate messages are also easily dealt with. If the sequence number of a message received has already been acknowledged, the received message is considered a duplicate and is ignored.

3. Connection termination phase: A connection is terminated using four packets: two packets that indicate the termination requests of each side and two packets to acknowledge each of the termination requests.

In the sections to follow the three phases of the protocol will be discussed in more detail. Each section first describes the detail of a particular phase and then a discussion follows of how the phase is added to a transition system structured model.

3.3.2 A Transition System Specification for TCP

A TCP process interfaces on one side with user or application processes and on the other side to a lower level protocol such as the Internet Protocol. A minimum TCP user interface must

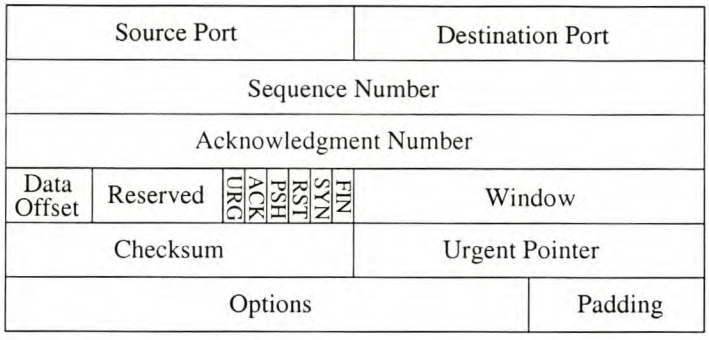


Figure 12: The format of the TCP segment header.

provide the following operations: *open*, *send*, *receive*, *close*, *status* and *abort*. The interface with the lower-level protocol is left unspecified, but calls such as *send* and *receive* are assumed to send and receive information over a network.

A message in the TCP protocol is called a segment. A segment consists of a header followed by the user data. The protocol uses sequence numbers for control flow and each data octet (a byte) in the segment is assigned a sequence number. The sequence number field in the header, shown in Figure 12, specifies the sequence number of the first data octet in the segment. Using the length of the segment, a cumulative acknowledgement mechanism is employed. The acknowledgement of sequence number X indicates that all octets up to but not including X have been received.

The sequence number range starts from a precomputed initial segment sequence number (ISN). This number is computed to prevent segments from one incarnation of a connection being used while the same sequence numbers may still be present in the network from an earlier incarnation. To ensure that a segment does not carry a sequence number which may be a duplicate of a previous connection, a TCP process must wait a maximum segment lifetime upon recovering from abnormal termination, before sequence numbers are assigned to segments. It is assumed that the maximum segment lifetime on a network will not exceed a few milliseconds.

A cycle through the possible values of a 32-bit variable is used for the sequence number range—referring to both the acknowledgement number and the sequence number shown in Figure 12. This means that a connection with a transmission speed of 100 megabits/second will take 5.4 minutes to use all of the 2^{32} octets of sequence number range. The large sequence range ensures that a sequence number is not re-used before it has been acknowledged because the maximum segment lifetime is only a few milliseconds.

Additional control information is passed between two communicating TCP processes by means of the control bits in the header. The bits are defined as:

- URG: This bit indicates to the receiving user process that in some point further along in the data stream that the receiver is currently reading there is urgent data. On receiving this bit, the process should take action to handle the urgent data as quickly as possible.
- ACK: A segment containing this bit indicates that the sending TCP process is acknowledging received data.
- PSH: Usually a TCP process will wait for enough data to fill the user data buffer, but when this bit is set all data received should immediately be pushed through to the receiving user.
- RST: If a TCP process receives this bit it should delete all the information currently associated with the connection and close the connection.
- SYN: If this bit is set it means that the information in the received packet should be used to initialise a new connection.
- FIN: This bit indicates that there is no more data coming from the sender TCP process.

3.3.3 Identifying states, actions and events

The informal specification [9] describes the TCP protocol as a state machine. Figure 13 shows a reachability graph for the protocol. Eleven states and ten events are identified in the specification. The states are:

- CLOSED - There exists no connection.
- LISTEN - A TCP process is waiting for a connection request from any remote TCP process.
- SYN_SENT - A TCP process is waiting for a matching connection request after having sent a connection request.
- SYN_RECEIVED - A TCP process is waiting for an acknowledge on a connection request after having both received and sent a connection request.
- ESTABLISHED - There exists an open connection between the two TCP processes. This is the normal state for data transfer.

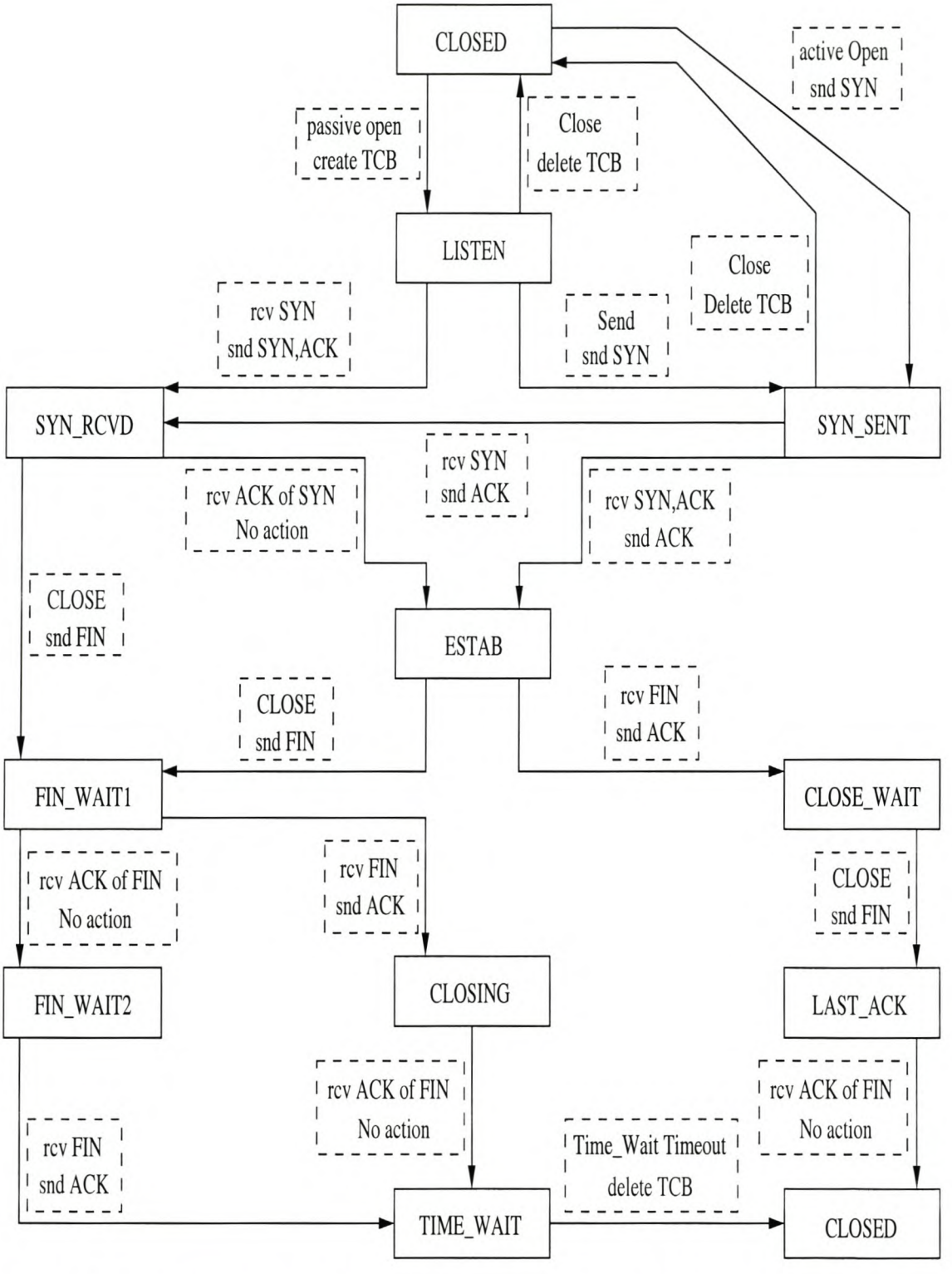


Figure 13: A reachability graph for the TCP protocol. The solid boxes represent the states. The top line of the dotted line boxes indicates the event and the bottom line the action associated with the event.

- FIN_WAIT1 - A TCP process is waiting for a connection termination request from a remote TCP process, or an acknowledge of the connection termination request previously sent.
- FIN_WAIT2 - A TCP process is waiting for a connection termination request from a remote TCP process.
- CLOSE_WAIT - A TCP process is waiting for a connection termination request from a local user.
- CLOSING - A TCP process is waiting for a connection termination request acknowledgement from a remote TCP process.
- LAST_ACK - A TCP process is waiting for an acknowledgement of the connection termination request previously sent to a remote TCP process.
- TIME_WAIT - A TCP process is waiting for sufficient time to pass to allow for a remote TCP process to receive the acknowledgement of its connection termination request.

A TCP process progresses through the states mentioned above in response to the following events. The events are grouped into three parts:

1. User Calls
    - *Active Open* or *Passive Open*: A user specifies an *Active Open* call if the user wants to synchronise immediately with another TCP process. A *Passive Open* call allows for a TCP process to listen for any incoming connections.
    - *Send*: The *Send* call causes the data contained in a user buffer to be sent.
    - *Receive*: The *Receive* call passes available data received from a sender to a user.
    - *Close*: This call causes a TCP process to end a connection.
    - *Abort*: This call causes a TCP process to abort a connection.
    - *Status*: A TCP process will return information on the current status of the connection if a user specifies a *Status* call.
2. Arriving Segments
    - *Segment_Arrives*: This event indicates the arrival of a packet. Several tests are performed on a packet to test its correctness. For example, the packet's checksum is computed, its sequence number is compared against the current receiving window

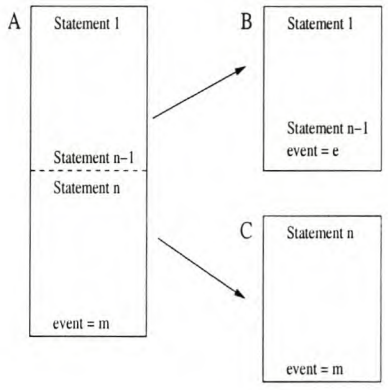


Figure 14: An action **A** fragmented into actions **B** and **C** by adding event *e*.

and the control bits included in the packet's header are evaluated to determine the current state of the sender TCP process.

3. Timeouts

   - *User_Timeouts*: A user can specify a time limit on the delivery of data to the destination.
   - *Retransmission_Timeout*: Each packet should be acknowledged before the *Retransmission_Timeout* otherwise the packet is resent.
   - *Time-Wait_Timeout*: After both TCP processes have sent their final packets the processes must wait the period of the *Time-Wait_Timeout* before the connection can enter the closed state. This is to ensure that a TCP process can retransmit its final packet in case the packet gets lost.

It is possible to implement a TCP protocol structured as a transition system using only the states, events and actions identified in [9]. However, some of the actions will be too complex, consequently complicating the testing phase. The actions that were identified as being too large were divided into smaller fragments. An action is divided into fragments by adding extra events to the specification. For example, Figure 14 shows how an action **A** is fragmented into actions **B** and **C** by adding event *e*. A total number of 21 events resulted from fragmenting actions that were too complex and a transition table consisting of 186 entries was defined.

3.3.4 Constructing the Model

As mentioned, the behaviour of TCP can be divided into three phases: the connection establishment phase, the data transfer phase and the connection termination phase. Each of these

phases can be studied independently. In our approach of constructing a transition system structured model, each of these three phases was added separately. But before these phases are described, the global data structures of the protocol are defined.

Global Variables and Data Structures

Information on a connection is kept within a data structure called the transmission control block (TCB). The TCB holds information such as the initial segment sequence number (ISN), the initial received segment sequence number (IRS), the window size and other information required to control the behaviour of the protocol. Information about a connection is transmitted between two TCP processes by means of the TCP header that is included in each segment. The necessary information needed to fill a TCB is derived from the data in each segment header. The TCB structure was implemented using the Promela **typedef** definition (Shown in Figure 15). All the variables were declared as bytes to keep the state vector as small as possible.

The final model consists of five processes: two user processes, each communicating with a TCP process, and two TCP processes which in turn are connected to an IP process. Eight message channels were declared: two for the communication between a user process and a TCP process and two for the communication between a TCP process and the IP process. The message type of a channel declared for communication between a TCP process and the IP process represents a TCP segment. It is declared as:

```
chan TCP1toIP = [0] of { byte, byte, byte, byte, byte };
```

The first four bytes of the message type represent the segment header and the final byte represents the data buffer. Only four fields of the TCP header are included in the model: the sequence number, the acknowledgement number, the control flags and the window field. As a first experiment we have decided to implement only a basic implementation of TCP and the urgent pointer and options field were not included in the model. With the global variables and structures defined, the behaviour of the three phases can be added to the model.

```
1  typedef TCBType {
2      byte ISS, SND_UNA, SND_NXT, SND_WND;
3      byte IRS, RCV_NXT;
4      byte SEG_SEQ, SEG_ACK, SEG_WND;
5      byte SEG_LEN, SND_WL2, SND_WL1;
6      byte SEG_CTL;
7      byte Smss, Rmss, Rcnt, Scnt;
8      byte Rfst, Rlst, Sfst, Slst;
9      byte Cwin;
10     byte Sbuf [BufSize];
11     byte Rbuf [BufSize];
12 }
```

Figure 15: The TCB record structure of TCP implemented in Promela using a **typedef** definition.

Connection Establishment

A connection between two TCP processes can be established as a result of two different scenarios. The first scenario is when two users wanting to communicate both issue an *Active Open* call (The difference between an *Active Open* and a *Passive Open* call was described in Section 3.3.3). Synchronisation packets are sent immediately and if no errors occur a connection is established. The second is when one user issues an *Active Open* call after the other has issued a *Passive Open* call. This type of connection establishment describes a typical client/server relationship: The server issues a *Passive Open* call waiting for an incoming request of a client who has issued an *Active Open* call.

The following example illustrates how a client TCP process (**TCP_A**) and a server TCP process (**TCP_B**) establish a connection where the user at the client side issued an *Active Open* call and the user at the server side issued a *Passive Open* call. The two TCP processes advance through the states of the reachability graph shown in Figure 13. It is also assumed that no errors occur during the connection setup.

The two TCP processes start with **TCP_A** in the CLOSED state and **TCP_B** in the LISTEN state. **TCP_B** is already in the LISTEN state because the user at the server side issued a *Passive Open* call.

When the user at the client side issues an *Active Open* call, **TCP_A** will send a synchronisation packet and it will enter the SYN_SENT state (Sending a synchronisation packet means that the SYN bit is set in the segment header). On receiving this packet **TCP_B** sends an acknowledge of the synchronisation packet (SYN and ACK bits set) and enters the SYN_RECEIVED state. If **TCP_A** receives the packet sent by **TCP_B** it enters the ESTABLISHED state. The user on the client side can now send data and **TCP_B** will enter the ESTABLISHED state on receiving the next packet sent. The connection is now in the

ESTABLISHED state and the "three-way handshake" is completed.

Two important design issues for the model are identified in the connection establishment phase:

1. **The communication between a user process and a TCP process:** The channel declaration shown below is used for the communication between the mentioned processes.

   chan U1toTCP1 = [0] of { **byte**, **byte** };

   The first byte of the message type is used for the commands that a user may issue to a TCP process, and it is also used for the result that a TCP process returns on a command issued to it. The second byte is used for user data from a user process to a TCP process and vice versa. It was chosen to model the communication between the two processes such that when a user process issues a command to a TCP process it will block until the TCP process replies.

2. **Recording of events in a TCP process:** The outline of a TCP process is shown in Figure 16. The outer **do...od** construct is used as an infinite loop and the inner **do...od** construct is used to combine the interpreter and the transition table. The **if...fi** construct is used to record external events such as: user commands (Lines 4 to 5), arriving segments (Lines 6 to 8), pending requests (Lines 9 to 13) and retransmission timeouts (Lines 14 to 15).

Data Transfer

Once a connection has been established, user data can be transmitted between the two sides. The major issue of this stage is sequence numbers. The following concepts relate to sequence numbers:

- **The receive window size:** The TCP protocol uses a form of flow control called a sliding window protocol. The sliding window protocol can be visualised as shown in Figure 17. The numbers 1 through 11 represent data octets. The receiving TCP process advertises a number that indicates how many data octets it is willing to accept. This number is called the *offered window* and it depends on the availability of user buffer space at the receiver side.

```
1  do
2  :: (1) →
3      if
4      :: UserIn?Event, UData →
5          skip /* Test if user made a call */
6      :: IPin?TCB.SEG_SEQ, TCB.SEG_ACK,
7              TCB.SEG_CTL, TCB.SEG_WND, IPData →
8          Event = SegmentArrives
9      :: (CloseCallQ == TRUE) && (State == ESTABLISHED) →
10         Event = CloseCall;
11         CloseCallQ = FALSE
12     :: (UserWaitsData == TRUE) && (TCB.Rcnt > 0) →
13         Event = ReturnData2User
14     :: timeout →
15         Event = RetransTimeout
16     fi;
17     do
18     :: (State == CLOSED) && (Event == OpenCallP) →
19         State = LISTEN;
20         Action0()
21     :: (State == LISTEN) && (Event == OpenCallP) →
22         /* State = LISTEN; */
23         Action4()
                 ⋮
24     :: else →
25         break
26     od
27 od
```

Figure 16: The outline of a TCP process implemented in Promela.

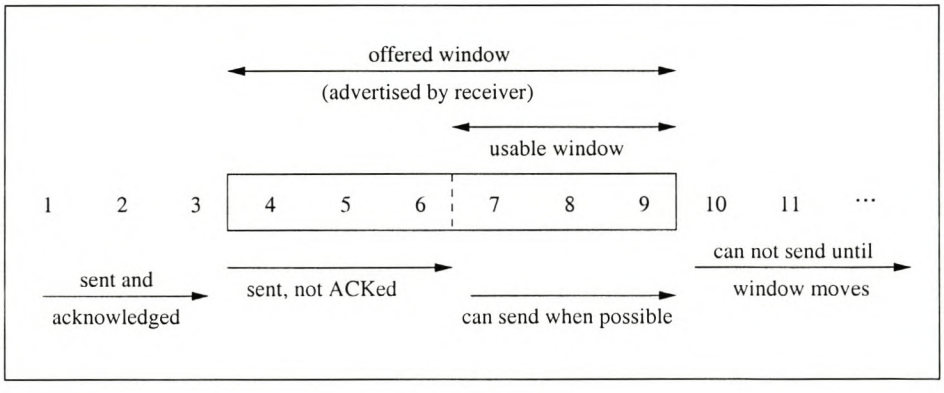


Figure 17: Visualisation of the TCP sliding window protocol.

In this example the *offered window* covers octets 4 through 9, which means that the receiver process has acknowledged all octets up to and including number 3, and has advertised a window size of 6—the window size is relative to the acknowledged sequence number. The sender process computes its *usable window* before each send command is issued to the underlying protocol. Over time the sliding window moves over higher ordered sequence numbers as the receiver process acknowledges data [47].

- **Retransmission of data:** One of the reliability methods of TCP is the acknowledgement of data octets sent: a receiving TCP process must acknowledge the data that was sent from a sending TCP process. However, segments sent can get lost. The TCP protocol recovers from segments lost by using a timer to force the retransmission of segments.

  Fundamental to the retransmission timeout of TCP is the measurement of the round-trip time (RTT). The RTT is the time it takes for an octet to go from a sending TCP process to a receiving TCP process and back. The retransmission timeout is typically computed using the RTT in an algorithm specified by Karn [36].

- **Old or duplicate segments in the network:** These type of segments are easily dealt with in the protocol. A TCP process keeps track of the sequence number that it expects next using the variable $RCV.NXT$. If a connection is established with an IRS of a, $RCV.NXT$ is set to $a+1$. If the segment arrives with sequence number equal to $a+1$ and the length of the segment is b, $RCV.NXT$ is set to $RCV.NXT+b$. A TCP process only accepts the octets in a segment that has a sequence number greater or equal than its $RCV.NXT$ value.

The send and receive buffers of a TCP process are implemented in the model as arrays of bytes with two elements each (Shown in lines 10 and 11 of Figure 15). Using buffers with only two elements keeps the model tractable while allowing for the verification of the various control flow concepts.

The Promela **timeout** statement becomes enabled only when every other statement in the system is blocked. This property of the **timeout** statement can be used for an elegant implementation of the retransmission timeout in a model structured as a transition system. For example, suppose the IP process chooses to lose a packet sent by a TCP process. All the processes in the model will then become blocked: the two TCP processes will wait for input from each other, the two user processes will wait for a reply from the TCP processes and the IP processes will wait for input from any of the two TCP processes. The **timeout** statement

will then become executable and the **Event** variable will be set to indicate a retransmission timeout (Shown in lines 14 and 15 of Figure 16).

Connection Termination

A connection is terminated when one of the users issue a *Close* call. To successfully terminate a connection, four segments are required: two segments that indicate the termination requests of each side and two segments to acknowledge each of the termination requests.

Continuing with the example sketched in the connection establishment stage, the steps to terminate a connection are as follows: The user at the client side starts the termination phase by specifying a *Close* call. Upon receiving this call **TCP_A** will send a finishing packet (FIN bit is set) and enters the FIN_WAIT state. **TCP_B** enters the CLOSE_WAIT state on receiving the packet and sends two packets in succession. First an acknowledge is sent for the finishing packet and then **TCP_B** sends its own finishing packet. On receiving the first packet from **TCP_B**, **TCP_A** enters the FIN_WAIT2 state and when the second packet is received, **TCP_A** enters the TIME_WAIT state. **TCP_A** sends a final packet to acknowledge the finishing packet of **TCP_B** and enters the CLOSED state. On receiving this packet **TCP_B** also enters the CLOSED state and the connection is closed.

As with any protocol, abnormal termination is a possibility. The TCP protocol recovers from abnormal termination by using a timeout that is specified by the user in an *Open* call. If data are not successfully delivered to the destination within the timeout period, the connection is aborted.

Similar approaches to those used in the previous stages were used to add the connection termination stage to the model. A final important feature regarding the memory issues of a transition system structured model is the use of the Promela **d_step** feature. This feature is used to execute the statements within its scope in one indivisible step. Its use reduces the number of possible interleavings for the statements of a model, therefore causing a reduction in the state space and ultimately a reduction in memory use. The statements within the scope of a **d_step** must be deterministic, may not jump to labels outside its scope and statements other than the first may not block.

The **d_step** feature can be used in most of the actions of a transition system structured model, except for those that contain message passing statements (these statements may block). It is however possible to use the **d_step** feature such that the message passing statements are excluded as shown in Figure 18.

```
#define ReturnToUser() \
    UserOut!Event, UData; \
    d_step { \
        UData = NoAction; \
        Event = NoEvent \
    }
```

Figure 18: An Action of the TCP model implemented in Promela with the **d_step** feature added.

Two different models—where the **d_step** feature was used in one of the models only—were verified using the same LTL formula. SPIN used 50 Megabytes to verify the model with the **d_step** feature added compared to 64 Megabytes to verify the model without the feature added.

3.3.5 Simulating the TCP Model

Using SPIN in simulation mode, each phase of the protocol was debugged before the next was added. This method revealed trivial errors and gave insight into the model. For example, the simulator was used to track the values of the **State** variables of the two TCP processes as they progressed through the different phases.

Figure 19 shows a window containing a message sequence chart and a window that displays the values of the global and local variables for each process at the end of a simulation run. It is a simple exercise to locate the action which executed incorrectly if a run ends with the two TCP processes not in the correct states.

The output shown in Figure 19 shows that both the TCP processes ended in the ESTABLISHED state—the bold black arrows indicate the **State** variables of the two TCP processes. This is the desired result for the connection establishment phase. With the model in this form, the next phase can be added.

3.3.6 Verifying the TCP Model

Holzmann suggested in [24] that the first step of the verification process should be to identify correctness properties that are relevant and require formal proof. This should also be the first step when following the transition system approach. Holzmann then suggests that an executable abstraction should be constructed that has enough expressive power to capture the essence of the solution, and no more. With the transition system approach one of the goals is to find the correct structure for a system and a model structured accordingly could

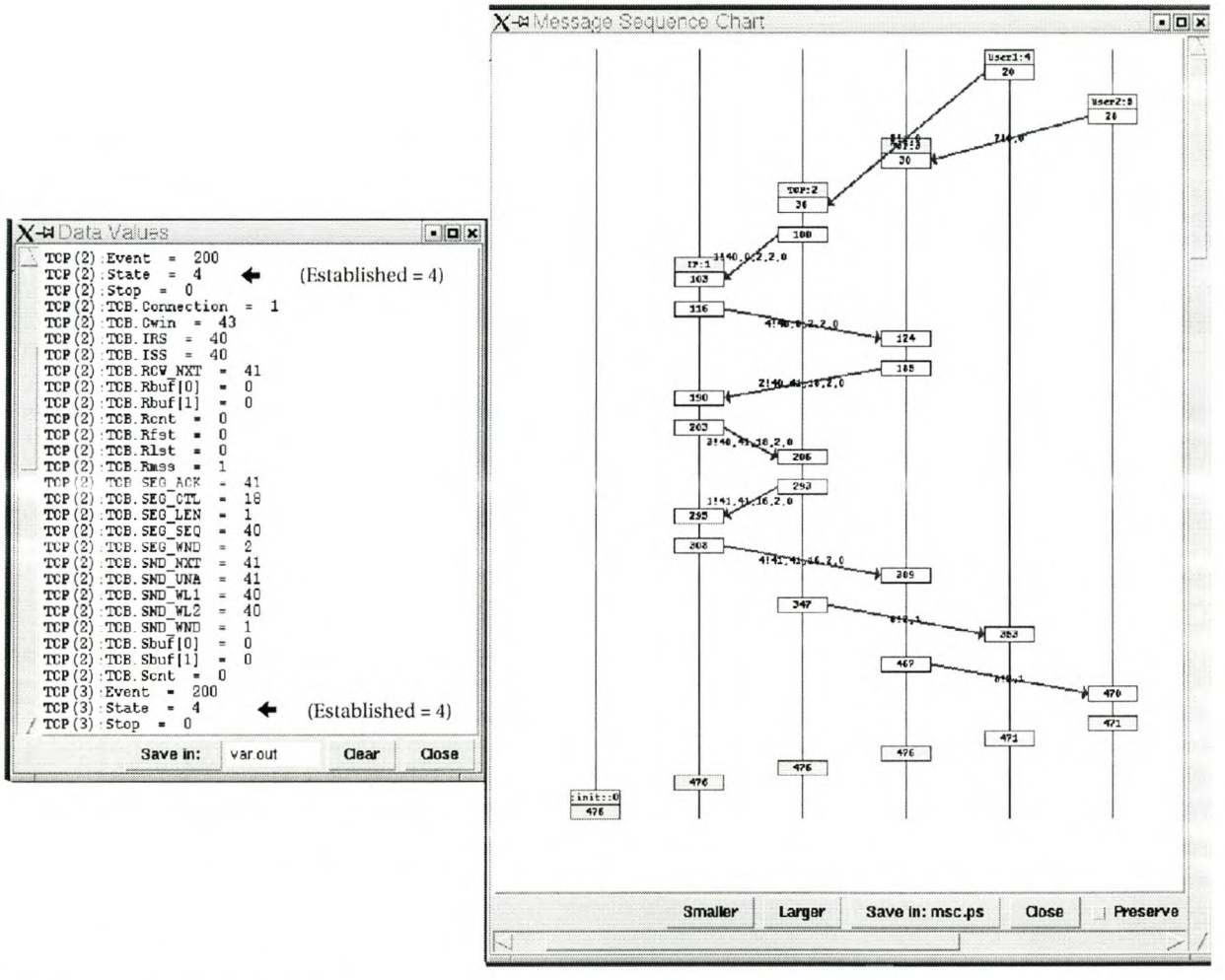


Figure 19: Two windows that show the message sequence chart (right) and the local and global variables for all processes (left) of a simulation run of the TCP protocol implemented in Promela.

have more expressive power than required for the identified properties.

In this experiment the two user processes were implemented in a client/server relationship. The client process first connects to the server process after which the client process sends a series of data packets to the server. The client process then closes the connection. This setup encapsulates the three stages of the protocol.

The first and simplest property verified was to check for the absence of deadlock. Spin used 36MB to show that the model is deadlock free. The modest memory usage results from the structure of the model that limits the number of possible paths. Only actions set the **Event** variable and the actions executed depend on the environment constructed in the model. Also, actions are atomic and testing for the occurrence of new external events only occurs after the completion of an action, therefore reducing the number of possible interleavings. However, a model that is deadlock free can still contain safety and liveness errors. For this experiment we used a liveness property, which is discussed next, to determine whether all three phases of the protocol complete correctly.

The auxiliary variable **U1End** is used to indicate the completion of the client user process (Shown in Lines 3 and 14 of Figure 21). This variable is used in the LTL formula: $\Diamond p$, where p is defined as:

#define p U1End == TRUE.

The symbol "$\Diamond$" represents the temporal operator "finally". Spin revealed a subtle error when this property was verified: If the client TCP process sends a synchronisation segment before the server TCP process has entered the LISTEN state, the connection will livelock. The error trace is shown in the message sequence chart of Figure 20. The first step in correcting the error was to locate only those actions that form part of the error trace. This is easily done by following the trace of the **State** and **Event** variables in the simulation output shown in the window on the left side of Figure 20.

Studying the selected code, it was found that the action responsible for retransmission contained the error: When a TCP process is in the CLOSED state and a segment is retransmitted, only the SYN bit should be set. In all other states the SYN and ACK bits should be set. The ACK bit should not be set during the retransmission of a packet in the CLOSED state because the packet selected for retransmission is the first packet that was sent to establish a connection. Clearly this packet cannot acknowledge any received packets because no packets

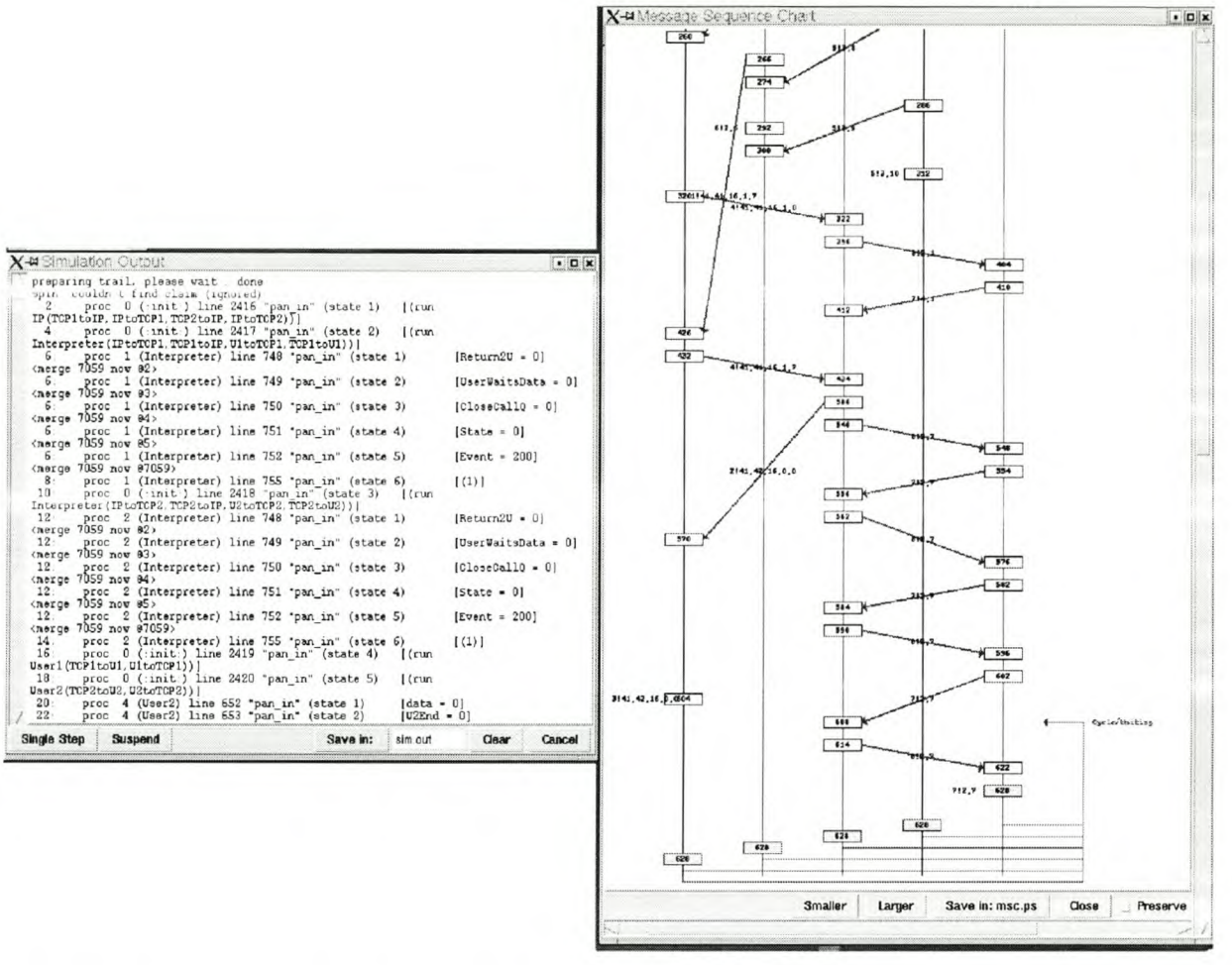


Figure 20: Two windows that show the message sequence chart (right) and the simulation output (left) of an error trace produced by Spin.

have yet been received. In our implementation the ACK bit was always included. This error would have been difficult to detect with conventional testing methods.

3.3.7 Deriving an implementation

The translation from the TCP model into C consisted of three main steps: The implementation of the interpreter, the translation of the transition table and the translation of the actions. In this example the interpreter and the transition table were combined using if- and switch-statements, maintaining the same structure as used in the Promela model. The code fragment showed in Figure 22 illustrates this technique.

Figure 23 shows how the model code that is used to test for new events (top half of the figure)

```
1  proctype User1(chan TCPin, TCPout)
2  {
           ⋮
3      U1End = FALSE;
4      Event = OpenCallA;
5      TCPout!Event, data;
6      TCPin?Event, data;
7      data = 7;
8      do
9      :: data < 11 →
10         Event = SendCall;
           ⋮
11     Event = CloseCall;
12     TCPout!Event, data;
13     TCPin?Event, data;
14     U1End = TRUE
15 }
```

Figure 21: A user process in the TCP Promela model.

```
do
:: TRUE →
    if
    :: ( expression1 ) →
        ...
    :: ( expression2 ) →
        ⋮
    fi
    do
    :: (Sate == StateValue1) && (Event == EventValue1) →
        ...
    :: (Sate == StateValue1) && (Event == EventValue2) →
        ⋮
    od
od

    ————————————————————

while (TRUE) do {
    if ( expression1 ) {
        ...
    else
        if ( expression2 ) {
            ⋮
    }
    switch ( State ) {
        case StateValue1:
            switch ( Event ) {
                case EventValue1:
                ...
                case EventValue2:
                ⋮
                }
            ⋮
    } }
```

Figure 22: The translation of the Promela **if...fi** and **do...od** constructs into equivalent C code.

```
1  do
2  :: (1) →
3      if
4      :: timeout →
5          Event = RetransTimeout
6      :: IPin?TCB.SEG_SEQ, TCB.SEG_ACK,
7              TCB.SEG_CTL, TCB.SEG_WND, IPData →
8          Event = SegmentArrives
9      :: (UserWaitsData == TRUE) && (TCB.Rcnt > 0) →
10         Event = ReturnData2User
           .
           .
           .
11     fi;
       ------------------------------

   while ( TRUE ) {
12     if ( RTimerB == TRUE )
13         Event = RetransTimeout;
14     else {
15         if ( ( PortIsSet == TRUE ) && ( (n = ReceiveBuf(TCB[C] → Sockfd)) != -1)
16             && ( (BufSize - TCB[C] → Rcnt) > n ) )
17             Event = SegmentArrives;
18         else {
19             if ( (UserWaitsData == TRUE) && (TCB[C] →Rcnt > 0) )
20                 Event = ReturnData2User;
                   .
                   .
                   .
21     }
           .
           .
           .
22 }
```

Figure 23: The Promela code that is used to test for the occurrence of new events (top-half) translated to equivalent C code (bottom-half).

is translated into equivalent C code (bottom half of the figure). The three guards shown in lines 4, 6 and 9 are used to test for the occurrence of a retransmission timeout, the arrival of a segment or whether a user is waiting for data.

The transition table was implemented using nested switch-statements, one to compare the *Event* variable and others to compare the *State* variable associated with each event. This method of implementation was used because the average size of an action is rather large and therefore the overhead added by the interpreter to the execution time of an action was considered negligible. Also the close relation between the Promela **do...od** construct and the switch-statement of C simplified translation. Figure 24 shows an excerpt of the transition table implemented in Promela (top half of the figure) and the translated C code (bottom half of the figure). Lines 6 and 19 are commented out for efficiency reasons (The state does not change).

It is a simple exercise to derive the code for the actions following the code of the model as guideline. Additional code was added for the manipulation of data-buffers and detail regarding the environment. Figure 25 shows how an action in the model (top half of the

```
1  do
2  :: (State == CLOSED) && (Event == OpenCallP) →
3      State = LISTEN;
4      Action0()
5  :: (State == LISTEN) && (Event == OpenCallP) →
6      /* State = LISTEN; */
7      Action4()
8  :: (State == SYNSENT) && (Event == OpenCallP) →
           ⋮
9  :: (State == CLOSED) && (Event == OpenCallA) →
           ⋮

   ──────────────────────────────────────────

10 while( Event != NoEvent ) {
11     switch ( Event ) {
12         case OpenCallP :
13             switch ( State ) {
14                 case CLOSED :
15                     State = LISTEN;
16                     Action0();
17                     break;
18                 case LISTEN :
19                     /* State = LISTEN; */
20                     Action4();
21                     break;
22                 case SYNSENT :
                       ⋮
23             }
24             break;
25         case OpenCallA :
26             switch ( State ) {
27                 case CLOSED :
                       ⋮
```

Figure 24: An excerpt of the transition table of the TCP protocol implemented in Promela (top-half) and the translated C code (bottom-half).

```
1  #define Action11() \
2      d_step { \
3          if \
4          :: UserWaitsData == 0 → \
5              UserWaitsData = 1; \
6              Event = NoEvent \
7          :: else → \
8              UInfo = INSRES; (* INSufficient RESources *) \
9              Event = ReturnToUser \
10         fi \
11     }

       ─────────────────────────────────────────────

12 void Action11() {

13     if ( UserWaitsData == 0 ) {
14         UserWaitsData = 1;
15         Event = NoEvent;
16     }
17     else {
18         SRBuf→Result = MyTCP_Error;
19         SRBuf→ErrorNum = INSRESC; (* INSufficient RESources *)
20         Event = ReturnToUser;
21     }
22 }
```

Figure 25: An action of the TCP protocol implemented in C. The C-code in the bottom-half is derived from the Promela code in the top-half.

figure) corresponds to an action in the implementation (bottom half of the figure). The action shown is used to queue a receive request from the user process if no data are available. Only one receive request is queued (Lines 4 to 6). If the user process issues a second request an error message is returned (Lines 7 to 9). The code in the implementation is similar to the code in the model. In the model the variable *UInfo* is returned to the user process (Line 8) and in the implementation the structured type *SRBuf* is returned (Line 18 to 19).

Procedures were used to implement actions in the implementation. This simplifies the code used to perform low-level operations. Also, the relative sizes of the actions were considered large enough such that the addition of the overhead of a procedure call would not have an huge influence on performance.

The errors discovered during the implementation phase were all related to data manipulation and interfacing with IP. This means that the errors were confined to detail within the actions which should be detected in the testing phase, as explained in Section 3.2.

Performance evaluation

A simple file transfer utility was implemented to test the TCP implementation. Files were sent from one computer to another on a subnet using only a hub to connect the two computers. Small files, less than 5KB were successfully transmitted with no noticeable difference in transfer rate compared to the Linux TCP implementation. However, for larger files the transfer rate was unacceptable. For example, the average transfer rate for a 100MB file was 18kilobits per second. The same file transfered with the Linux TCP implementation has an average transfer rate of 930kilobits per second.

The main reason for the slow transfer rate is that data buffers are copied unnecessarily. The transition system structured TCP was implemented as a user process on Linux. This process interfaces with the Linux IP process on the one side and with a user process on the other side. Data buffers are copied from the IP process to the TCP process and then to the user process. While copying takes place between the TCP process and the user process, no IP packets are accepted by the TCP process. This causes the receiver process to drop packets and the sender process to retransmit. The high retransmission rate consumes a large percentage of the bandwidth.

The problem could be alleviated by either implementing the TCP process as part of the kernel protocol stack, removing the copying of data between processes [41, Chapter 9], or using separate threads to handle insertions and removal from the TCP buffer. However, an optimal implementation was not the goal of this experiment.

3.4 Summary

In this chapter the technique to structure both a model and an implementation of a protocol as a transition system was discussed. The technique was illustrated using the Alternating Bit protocol and the more complex TCP protocol.

Embedded systems that are mass produced usually have little memory and relatively slow processors to minimise costs. Assembly language—which is usually considered too low level (impractical) for development—is then considered to optimise both memory use and speed.

The Alternating Bit example showed that the transition system approach can be used to implement small protocols in assembly language. The technique improves the reliability and maintainability of assembly code for several reasons:

- Flow of control is verified at code level.
- Actions that are small, independent code fragments can be tested easily and thoroughly.

It is also possible to automatically translate the outline of a verified model into an implementation, leaving only the detail of the actions for manual coding. Furthermore, this approach combines the design and verification phases. There is no need to build a separate model—which often requires a considerable amount of time and skill to construct—to verify a design. The time spent on the design phase now also includes verification. The time spent to construct a model is often used as a criticism against model checking.

It was also shown that the technique scales for larger systems and the TCP protocol was used for this illustration. The same benefits that were identified for the Alternating Bit example in Section 3.2.1 also apply to the TCP example.

With the transition system approach, restriction on the structure of a design is used to couple verification and implementation more closely. The gain in correctness, however, leads to a decrease in efficiency—because of the overhead of the interpreter. The major drawback, however, is that not all systems are structured as transition systems. In the next chapter the automatic derivation of models from arbitrarily structured implementations will be discussed.

Chapter 4

Automated Model Extraction

Transition systems provide a rigorous basis for the translation of verified models into implementation code. The reverse—the derivation of models from implementation code—is also possible, as mentioned in Chapter 2. In fact, Holzmann showed that the process of extracting models from suitably structured C programs can be partially automated. However, not all implementations can be structured as transition systems because the overhead of interpretation is not acceptable.

The translation of arbitrarily structured Promela models into C code has been investigated by Löffler and Serhrouchni [46]. Unfortunately, there is an inherent problem with this approach: Models contain too little detail to allow derivation of efficient implementation code. In contrast, derivation of models from arbitrarily structured implementation code seems feasible because the removal of detail is simpler. What is needed is an effective mechanism for abstraction. We decided to investigate to what extent Holzmann's table-driven technique would also work for arbitrarily structured code written in a process-based implementation language. A process-based language was chosen to maintain a close similarity between the implementation and the modelling language. The idea is to retain the basic structure of an implementation and to replace irrelevant implementation detail by higher-level abstractions.

The language used for the experiment is called LF [51]. It has a similar structure to Promela and supports processes and message passing. In addition, essential operations needed to program at the hardware level are supported. The language will be introduced by describing an implementation of the alternating bit protocol. This example will then be used to illustrate application of the table-driven technique.

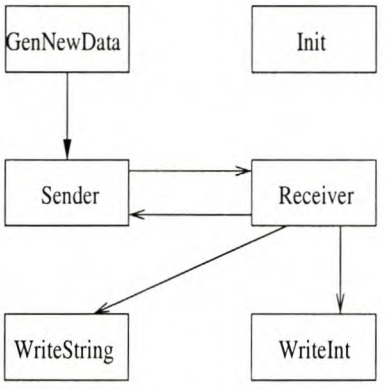


Figure 26: Communication between the processes of the Alternating Bit protocol implemented in LF.

```
1  PROCESS Sender(IN ackmsg : MsgChan; OUT datamsg : MsgChan; IN gendatamsg : MsgChan);
2  VAR
3      DataMsg : MsgType;
4      AckMsg : UINT8;
5      ControlBit : UINT8;
6  BEGIN
7      ControlBit := 0;
8      WHILE TRUE DO
9          gendatamsg ? data(DataMsg);
10         DataMsg.Bit := ControlBit;
11         REPEAT
12             datamsg ! data(DataMsg);
13             ackmsg ? ack(AckMsg)
14         UNTIL AckMsg = DataMsg.Bit;
15         ControlBit := 1 - ControlBit
16     END
17 END Sender;
```

Figure 27: LF code for the sender process of the alternating bit protocol.

4.1 Alternating bit protocol implemented in LF

The alternating bit protocol was described in Chapter 2. To keep the discussion as simple as possible, the protocol is implemented as cooperating processes on the same computer. The `Sender` process transmits data messages to the `Receiver` process which displays each message as it is received. The structure of the implementation is shown in Figure 26.

The `Sender` process is shown in Figure 27. Processes in LF communicate by sending messages over channels. Channels are created dynamically and can be passed to other processes via parameters. Three channel variables of the same type are passed to process `Sender` (line 1). Process `Sender` receives acknowledgements from the channel referenced by port variable *ackmsg*, sends data messages to the `Receiver` process via the channel referenced by port variable *datamsg* and receives new data from process `GenNewData` via the channel referenced by port variable *gendatamsg*. The reserved words IN and OUT are used to indicate

the direction of transfer. For example, process `Sender` is only allowed to send messages on the channel referenced by variable *datamsg*.

The three port variables in line 1 all reference channels of type *MsgChan*, which is declared as:

```
TYPE
   MsgChan = [data(MsgType), ack(UINT8)];
```

A channel can be used to transmit different types of messages and each type is associated with an alphabet symbol. In the declaration shown above, the alphabet symbol *ack* is associated with an unsigned 8 bit integer and the alphabet symbol *data* is associated with the user defined type *MsgType*. An alphabet symbol is used to indicate the type of a message in a communication command.

In lines 3 to 5 local variables *DataMsg*, *AckMsg* and *ControlBit* are declared. Variable *DataMsg* is used for receiving messages from process `GenNewData` and to send the data received from process `GenNewData` to process `Receiver`. Variable *AckMsg* is used for receiving acknowledgements from the `Receiver` process and variable *ControlBit* is used to store the sequence bit of the last packet sent.

The `Sender` process starts by initialising the control bit (line 7) and then enters an infinite loop, repeating the following steps: After receiving new data from process `GenNewData` (line 9), it sets the control bit of the new message and then sends the new message to the `Receiver` process. For each message sent, an acknowledgement is expected and if an error occurs, the message is retransmitted (lines 11 to 15). Finally, when the correct acknowledgement is received, the control bit is changed and the sequence of steps are repeated.

The standard CSP style notation for message passing is used in LF. For example, line 9 specifies that data should be received from the channel referenced by *gendatamsg* and placed into the variable *DataMsg* of type *MsgType*—which is associated with the alphabet symbol *data*.

The `Receiver` process is shown in Figure 28. This process receives messages sent by the `Sender` process via the channel referenced by variable *datamsg* and send acknowledgements via the channel referenced by variable *ackmsg* (line 1). The two local variables, *SenderMsg* and *ControlBit* are used to receive data sent by the `Sender` process and to test if a received message is valid.

```
1  PROCESS Receiver(IN datamsg : MsgChan; OUT ackmsg : MsgChan);
2  VAR
3      SenderMsg : MsgType;
4      ControlBit : UINT8;
5  BEGIN
6      ControlBit := 0;
7      WHILE TRUE DO
8          REPEAT
9              datamsg ? data(SenderMsg);
10             ackmsg ! ack(SenderMsg.Bit)
11         UNTIL SenderMsg.Bit = ControlBit;
12         ControlBit := 1 - ControlBit;
13         (* Display received data *)
14     END
15 END Receiver;
```

Figure 28: LF code for the receiver process of the alternating bit protocol.

```
1  PROCESS GenNewData(OUT datamsg : MsgChan);
2  VAR
3      NewDataMsg : MsgType;
4      Counter : INT32;
5      Data : UINT8;
6  BEGIN
7      WHILE TRUE DO
8          Counter := 1;
9          Data := 0;
10         WHILE Counter <= DataSize DO
11             NewDataMsg.Data[Counter] := Data;
12             Data := Data + 1;
13             Counter := Counter + 1
14         END;
15         datamsg ! data(NewDataMsg)
16     END;
17 END GenNewData;
```

Figure 29: LF code for the process in the alternating bit protocol used to generate new data.

The `Receiver` process also starts by initialising its control bit and then enters an endless loop. It keeps receiving and acknowledging messages until the control bit received is equal to its local control bit. If a message is accepted, the local control bit is changed and the received data displayed.

Three other processes that form part of the LF alternating bit protocol implementation are `GenNewData`, `WriteInt` and `WriteString`. Numerical values and strings are displayed using the two processes `WriteInt` and `WriteString`. Process `GenNewData` is used to generate new data. Its code is shown in Figure 29. The process repeatedly fills a data buffer with a series of integers and sends a message containing the buffer to the `Sender` process. Communication in LF is synchronous and the command in line 15 will block until process `Sender` accepts the message.

The five processes discussed above are instantiated in an initial process called `Init` shown in Figure 30. The command NEW is used to create various channels between processes (lines 7

```
1  PROCESS Init;
2  VAR
3      ckmsg : MsgChan;
4      datamsg : MsgChan;
5      gendatamsg : MsgChan;
6  BEGIN
7      NEW( datamsg );
8      NEW( ackmsg );
9      NEW( gendatamsg );
10     GenNewData( gendatamsg );
11     Sender( ackmsg, datamsg, gendatamsg );
12     Receiver( datamsg, ackmsg)
13 END Init;
```

Figure 30: LF code for the `Init` process of the alternating bit protocol.

to 9). The complete implementation is listed in Appendix B.

It should be noted that the packets sent between the `Sender` and the `Receiver` processes do not contain a message header or a checksum field. These two fields should be added in a real implementation but were left out for the sake of clarity.

4.2 From LF to Promela

Most constructs in LF can be translated directly to Promela. These include loops, alternative commands, most assignments, basic types and most structured types such as records and arrays. Since this subset sufficed to implement the alternating bit protocol, generation of a Promela model was an almost trivial exercise.

A number of abstractions can be applied to the alternating bit implementation which include the removal of processes `WriteString` and `WriteInt`—these processes are used only for output—and the simplification of the message type sent between the sender and receiver processes. The abstractions performed on the implementation code can be specified as a mapping from LF code to Promela code. For example, the processes `WriteString` and `WriteInt` map to empty Promela statements and all communication statements associated with these two processes map to the Promela *skip* statement.

The type of a data message is declared as:

```
TYPE
   MsgType = RECORD
      Data : ARRAY DataSize OF UINT8;
      Bit : UINT8
```

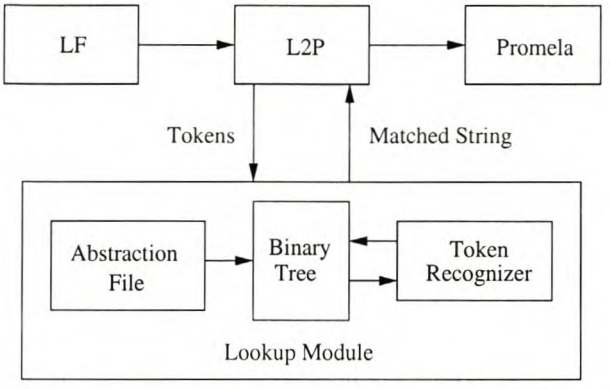


Figure 31: A schematic representation of the L2P translation tool.

```
END;
```

The record type consists of a total of 512 bytes: variable *Data* which is an array of 511 (*DataSize*) bytes and the 1 byte *Bit* variable. Only variable *Bit* is used for control flow and variable *Data* is abstracted to an array of two bytes. With a two byte buffer it is still possible to verify indexing. The code maps to:

```
#define DataSize 2

typedef MsgType {
  byte Data[DataSize];
  byte Bit
}
```

With manually defined mappings and the source code as input, it is in this case possible to construct a tool that automatically translates the LF source code to a tractable Promela model. Such a tool should consist of rules to translate LF code that need not be abstracted and a lookup mechanism to generate Promela code from the manually defined mappings.

Influenced by the success of the translation tool that Holzmann described in [28], we decided to construct a similar tool. Our tool, called L2P (LF to Promela), consists of a parser that contains simple translation rules and a lookup model which is used for the lookup mechanism. The structure of the tool is shown in Figure 31.

The lookup module consists of an abstraction file, a binary tree and a token recogniser. The LF part of a manually defined mapping is placed in the binary tree and the Promela part of

the mapping is placed in the abstraction file. A binary tree is used for pattern recognition because it simplifies the recognition of substrings. For each accepting path (string) in the binary tree there must be an entry in the abstraction file.

For example, Figure 32 shows how the constant declaration

$$DataSize = 511;$$

is placed in the binary tree and the Promela part,

$$\#define\ DataSize\ 2$$

in the abstraction file. The line of code listed in the top of the figure shows an input command to the tool. The code specified between the first pair of quotes is placed in the binary tree. The code is broken up into nodes as shown in the box labelled *Binary Tree*. Each node in the binary tree has a delimiter field. If a match is found in the tree, the delimiters in the first and last nodes of the match are used to find the corresponding Promela code in the abstraction file.

In this example the node with its token field equal to *DataSize* has a delimiter field equal to *PROMSTART1*. A delimiter name comprises of the prefix *PROMSTART* followed by an numerical value which is increased by one for each delimiter added. The node with its token field equal to ";" has a delimiter field equal to *PROMSTART2*. If the declaration

$$DataSize = 511;$$

is recognised in the input, the two delimiters will be used to substitute the Promela code that was automatically generated by the tool with the Promela code

$$\#define\ DataSize\ 2$$

The model generated for the alternating bit protocol is listed in Appendix C. This model was verified for the absence of deadlock and correct behaviour under packet loss. To verify correct behaviour under packet loss, the model was modified to include a lossy channel.

The success of this translation effort is due to the following:

1. There is a close correspondence between LF and Promela,
2. no low-level operations were used and
3. only a subset of LF was used.

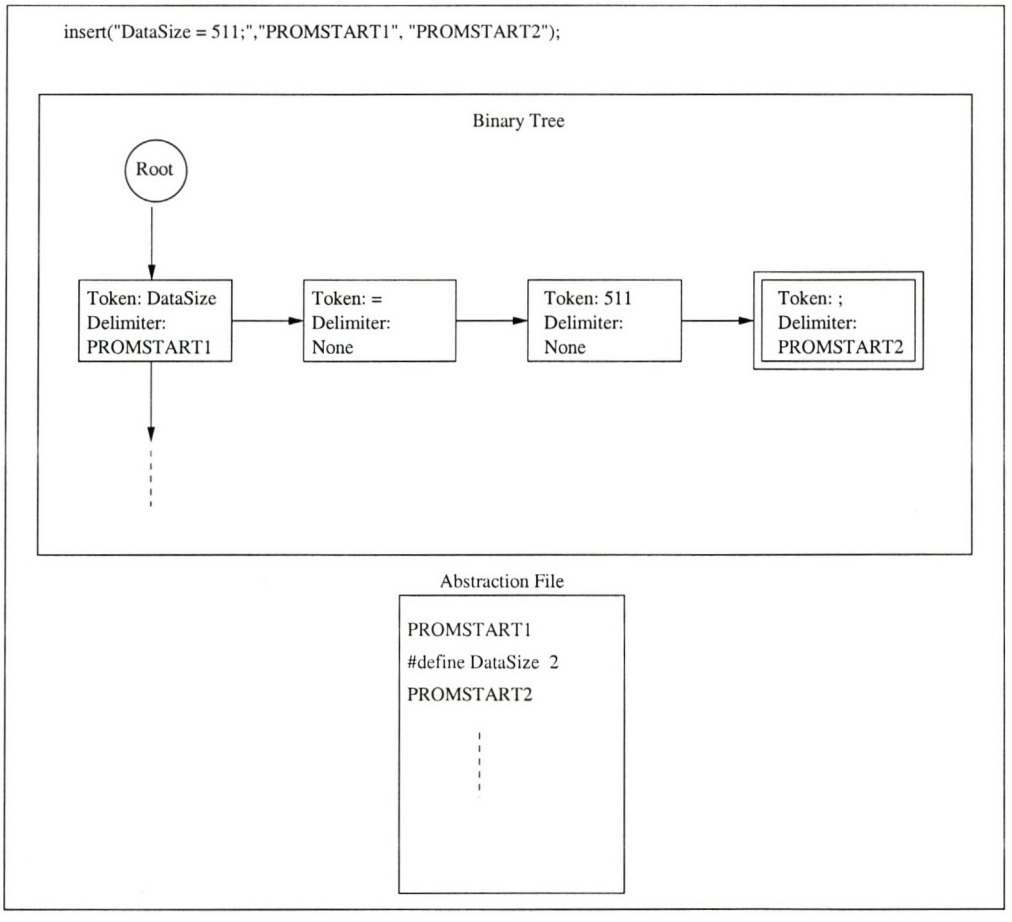


Figure 32: A schematic representation of the lookup module.

The TCP protocol was used again to investigate whether this technique would scale to verify more complex protocols.

4.3 Application to TCP

In general the following constructs in LF can be translated directly to Promela:

- **Assignments:** The syntax of an LF assignment is Pascal like and a Promela assignment is C like. It is only required to change the assignment symbol from ":=" to "=" during the translation of an assignment statement.
- **If-construct:** Unlike Pascal an IF-ELSIF-ELSE construct is supported in LF. The outline of this construct is shown below:

```
IF condition1 THEN
  statements1
ELSIF condition2 THEN
  statements2
ELSE
  statements3
END
```

The code is translated to Promela code as follows:

```
if
:: condition1 ->
  statements1
:: condition2 && !(condition1) ->
  statements2
:: !((condition1) && (condition2)) ->
  statements3
fi
```

- **Repetitive constructs:** The LF WHILE construct is also translated easily:

```
WHILE condition DO
  statements
END
```

This code is translated to:

```
do
:: condition ->
  statements
:: !(condition) ->
  break
od
```

LF also supports a REPEAT-UNTIL construct:

```
REPEAT
  statements
UNTIL condition
```

The REPEAT-UNTIL construct is translated to the Promela **do...od** construct:

```
bool auxiliaryVar = 0;

do
:: (condition) || (auxiliaryVar == 0) ->
  statements;
  auxiliaryVar = 1;
:: !((condition) || (auxiliaryVar == 0)) ->
  break
od
```

The auxiliary variable *auxiliaryVar* is used to ensure that the statements within the body of the construct are executed at least once, simulating the behaviour of the LF code.

- **Message passing:** As described in Section 4.1, LF channels are referenced via port variables. Different types of messages are associated with a channel type by means of alphabet symbols. Alphabet symbols are used to perform runtime type checking: Because multiple types of messages can be sent over the same channel, send and receive commands are matched using alphabet symbols. This approach of message passing is also followed by Joyce [5].

  In the code shown below a channel type channelType is declared with two alphabet symbols.

```
00  TYPE
01    channelType = [a(UINT32), b];

02  VAR
03    OUT port1 : channelType;
04    IN port2 : channelType;
05    var1 : UINT32;
06  BEGIN
07    port1 ! a(var1);
08    port1 ! b;
09    port2 ? b;
10    port2 ? a(var1);
```

The alphabet symbol a is associated with a 32-bit unsigned integer and alphabet symbol b has no type associated it. If an alphabet symbol has no type associated with it, it is called a signal. Signals are used for synchronisation and involve no copying of data. In lines 03 and 04 instances of *channelType* are declared. The equivalent Promela code is:

```
00  chan port1_a = [0] of {int};
01  chan port1_b = [0] of {bit};
02  chan port2_a = [0] of {int};
03  chan port2_b = [0] of {bit};
04  int var1;

05  port1_a ! var1;
06  port1_b ! 1;
07  port2_b ? 1;
08  port2_a ? var1;
```

Only one type of message can be associated with a Promela channel. It is therefore required to declare a Promela channel for each alphabet symbol in an LF channel type, shown in lines 00 to 03. Signals are translated to channels of type **bit** (Lines 01 and 03). Fields in a Promela communication statement can be constants. For the receive operation to be executable, the value of all message fields that are specified as constants must match the value of the corresponding fields of the message on the channel. This notion is used to translate LF communication statements that involve signals (Lines 06 and 07).

- **Process instantiation:** Instantiation of processes in LF and Promela is similar. The only difference is the keyword **run** that precedes the process name in Promela. For example, process DummyProcess is instantiated in LF as:

```
DummyProcess(var1, var2);
```

and in Promela as:

```
run DummyProcess(var1, var2);
```

- **Select construct:** LF supports a SELECT construct that has the same behaviour as the Promela **if...fi** construct. The outline of a SELECT construct is shown below:

```
SELECT guard1 THEN
  statements1
[] guard2 THEN
  statements2
END
```

This code is translated to:

```
if
:: guard1 ->
  statements1
:: guard2 ->
  statements2
fi
```

The LF SELECT construct also supports selective reception of messages. A selective receive is implemented as a guard in a SELECT construct that consists of a communication statement with an optional boolean expression. The boolean expression refers to the variables used in the communication. The following example illustrates this concept:

```
SELECT port1 ? a(var1) & (var1 > 0) THEN
  statements1
[] port2 ? a(var2) THEN
  statements2
END
```

The communication statement in the first line will only be executed if the value of the message on the channel is greater than zero. Because Promela does not support conditional reception direct translation is not possible. This feature could be simulated by an additional process but in general, this would increase the number of states generated significantly. Simulation can be achieved by adding an extra process that evaluates the contents of messages sent by a sender process. This process tags a message depending on its contents by adding a constant field to the message. The tagged message is then send to the receiver process which uses the tag (constant field) to select the appropriate code to execute.

- **Pointers:** This construct supported in LF cannot be translated directly to Promela. A restricted approach is to simulate a heap. It was showed in [18] how the dynamic creation of objects in Java can be simulated in Promela. An array of records is used to model the data area of a class. Each record in the array represents the data area of a created object. An index variable is used to indicate the next open slot. Visser et al. also followed this approach to translate C++ code to Promela [43].

  This approach can also be used to simulate pointers that point to data objects. An array of records must be constructed for each object type to which a pointer can refer. However, this approach increases the state space dramatically. LF supports type casting which means that a pointer can point to objects of different size. This additional complication is likely to render this approach impractical.

  In most cases it is best to abstract pointer usage rather than simulating it. The use of abstract data types facilitates pointer abstraction. The building blocks used in LF to implement abstract data types are processes. A queue is typically implemented as an abstract data type.

  Consider the example shown in Figure 33. Process Q is an abstract data type for a queue. Operations on the queue are performed by sending messages to process Q. Note that a new queue is created for each instance of the process. This is similar to multiple instances of an object.

  Lines 3 to 8 are the type definitions required. The type *Node* is the data type which is manipulated. The process Q provides two operations on the queue: Insertion is provided for by the first guard (Line 15) and removal is provided for in the second guard (Line 22). The initialisation of the queue is implicit as it takes place immediately after the instantiation of an instance of process Q (Line 13). Notice the selective receive used in the second guard to test if the queue is not empty (Line 22).

```
1  PROGRAM List;
2  TYPE
3      Node = POINTER TO NodeDesc;
4      NodeDesc = RECORD
5          next : Node;
6          b : UINT8
7      END;
8      QDesc = [ node(Node) ];
9  PROCESS Q( IN enQ : QDesc; OUT deQ : QDesc );
10 VAR
11     new,root : Node;
12 BEGIN
13     root := NIL;
14     WHILE TRUE DO
15         SELECT enQ ? node( new ) THEN (* insert operation *)
16             IF root = NIL THEN
17                 root := new
18             ELSE
19                 new.next := root;
20                 root := new
21             END
22         [] deQ ! node( root ) & root # NIL THEN (* remove operation *)
23             root := root^.next
24         END
25     END
26 END Q;
27 PROCESS Control;
28 VAR
29     i : UINT8;
30     n : Node;
31     OUT enQ : QDesc;
32     IN deQ : QDesc;
33 BEGIN
34     i := 0; NEW( enQ ); NEW( deQ );
35     Q( enQ,deQ );
36     WHILE i < 20 DO
37         NEW( n );
38         n^.b := i;
39         n^.next := NIL;
40         enQ ! node( n );
41         INC( i,1 )
42     END;
43     deQ ? node(n);
44     WHILE n # NIL DO
45         deQ ? node(n)
46     END
47 END Control;
48 BEGIN
49     Control
50 END List.
```

Figure 33: A queue implemented as an abstract data type in LF.

The LF implementation can be translated to Promela code as shown in Figure 34. The queue is represented as a circular buffer (Line 2). Variable *head* (Line 6) represents the *root* pointer variable in the LF code and -1 is used to represent a NULL pointer (Line 37). How the rest of the translation was done should be clear. Notice that the structure of the implementation also allowed for a simple and effective means to remove the selective receive used in the implementation code.

In the light of the preceding discussion direct translation seems feasible. An implementation of TCP in LF, written by an independent programmer, was selected to investigate whether this holds true for more complex protocols. The implementation consists of 820 lines of LF code and a total of 12 processes communicating over several different channels. The implementation is rather small compared to the UNIX implementation of the protocol which consists of more than 4000 lines of C code [48, Chapter 24]. The reason is that the LF implementation is a basic TCP implementation and does not support all the features of the UNIX implementation.

Despite the apparent similarity between LF and Promela, the translation was not successful for two reasons:

1. The SELECT construct with conditional reception was used extensively in the code.
2. Pointer usage was not confined to operations within abstract data types.

An investigation revealed that it would be possible to restructure the selected TCP implementation to support direct translation. This would entail the following:

- All complex data structures should be represented as abstract data types. Code structured accordingly facilitates translation because a complex data structure implemented as a process—which may contain pointer manipulation code—can easily be replaced by a more abstract version in a model. For example, the TCP data buffer can be implemented as an abstract data type. An outline of a possible implementation of such an abstract data type is illustrated below:

```
PROCESS Segment(
     (* Channels declared for communication to other processes *) );
VAR
  DataBuffer : ARRAY BuffSize OF CHAR;
```

```
1  #define MaxItems 30

2  short List[MaxItems];
3
4  proctype Q(chan enQ, deQ) {
5      short item;
6      short head, tail;

7      head = 0; tail = head + 1;
8      List[head] = -1;
9      do
10     :: enQ ? item →
11         if
12         :: tail != head →
13             List[tail - 1] = item;
14             tail = (tail % MaxItems) + 1
15         :: else
16         fi
17     :: deQ ! [head] →
18         List[head] = -1;
19         if
20         :: (head + 1) % MaxItems != tail →
21             head = (head + 1) % MaxItems
22         :: else
23         fi
24     od
25 }

26 proctype Control(chan deQ, enQ){
27     short i;

28     do
29     :: i < 20 →
30         enQ ! i;
31         i++
32     :: else →
33         break;
34     od;
35     i = 0;
36     do
37     :: i != -1 →
38         deQ ? i
39     :: else
40         break
41     od
42 }

43 init{
44     chan enQ = [0] of { int };
45     chan deQ = [0] of { int };

46     run Q(enQ, deQ);
47     run Control(deQ, enQ);
48 }
```

Figure 34: A model of the LF implemented queue shown in Figure 33.

```
  (* Other local variable *)
BEGIN
  WHILE TRUE DO
    SELECT (* Receive new data *) THEN
   ...
 [] (* Compute a checksum *) THEN
   ...
 [] (* Retrieve control bits *) THEN
   ...
    [] (* Retrieve sequence number *) THEN
   ...
    [] (* Retrieve acknowledgement number *) THEN
   ...
 (* other possible operations on data *)
 ...
 END
  END
END DataBuffer;
```

A process that uses the abstract data type issues commands to the *Segment* process and each of these commands is serviced by one of the operations in the SELECT construct. An option in the SELECT construct that proves too difficult for translation can now easily be substituted with functionally equivalent Promela code.

- Separate control flow from data manipulation. In Chapter 3 it was shown how the control flow of the TCP protocol can be implemented as a transition system. This can be implemented as a separate control process in LF. The control process interacts with other processes that are responsible for data manipulation.

Although it was considered unnecessary to develop a new implementation especially to support verification, it is obvious that this should be possible:

- Models of comparable complexity have been verified using SPIN. For example, the DEOS kernel [53] and the multiprocessor real-time kernel described in [6].
- In Chapter 3 it was shown that the control structure of TCP is simple enough to be verified by SPIN.

- The main problem with the original TCP implementation is the unrestricted use of pointers. This problem could be eliminated by restructuring the data manipulation code as discussed above.

The key to success seems to lie in the structure of software. In practice, however, most implementations are so complex that it is difficult to adhere to the stated guidelines. Pointers are often used in unrestricted ways, little use is made of abstraction and designs have no clear separation between data manipulation and control flow. Consequently, the goal of several research groups is to devise methods for supporting model extraction in general. For example, Visser et al. have developed a model checker targeted at verifying unrestricted Java code [52, 53]. Since abstraction is so important, program slicing, data abstraction and component restriction are used by the Bandera group to make model checking feasible for larger systems [8, 12]. Moreover, simple techniques such as encapsulating C statements to execute as one atomic block can be surprisingly effective [30]. Finally, the concept of a boolean program (only Boolean variables are allowed), and a model checker for Boolean programs, made it possible to verify device drivers consisting of several thousands of lines of C code [2].

Chapter 5

Comparison of Approaches

In this thesis two different strategies for using model checking in practice are compared: manual construction of models versus automated model extraction. The comparison was focused on protocols which are often used in embedded systems. For the manual approach the system was structured as a transition system. This was implemented by a simple interpreter, a transition table and small code fragments—called actions—associated with each transition. Automated model extraction was investigated by implementing a lookup table technique proposed by Holzmann [28].

It seems reasonable to expect that the quality of software will be influenced by different development methods. However, it cannot be concluded that one of the methods considered will always be preferable in practice. Therefore, the influence of each method on software quality should be judged by taking various important issues into account as discussed in what follows. In this way it is usually possible to select the best approach in specific circumstances.

Small embedded systems

Resources such as memory are limited in most embedded system environments. If execution speed is not of utmost importance, the transition system approach should be considered. If the amount of memory is severely limited, for example for simple devices targeted at the mass market, low-level coding is sometimes essential. Even in this case, it was shown in Chapter 3 how the transition system approach can be used to verify control flow directly and to simplify testing of data manipulation code. This strategy has the following advantages:

- Low-level coding can save a significant amount of memory and reduces the overhead inherent in interpretation. In this respect it is most important to represent data structures as compactly as possible, which is difficult to accomplish when using high-level languages.
- The structure imposed by a transition system allows direct verification of control flow.
- Code for data manipulation is clearly separated from the control flow code. Each data manipulation code fragment is isolated from the rest of the system in such a way that testing is simplified.

Complex protocols

In less restricted environments, where complex protocols are needed, it is advisable to use a high-level programming language to simplify code maintenance. The transition system approach is recommended if the overhead of interpretation is acceptable. It is possible to translate a verified model structured as a transition system into reasonable efficient implementation code by following the techniques described in Chapter 3. Most important is to avoid the overhead of copying messages and to structure the transition table in such a way as to optimise selection of the most important transitions. In addition, it helps to keep actions as simple as possible to simplify testing. The advantages of this approach are the same as discussed above although the overhead of interpretation can be significant.

Performance critical applications

Sometimes reasonably complex protocols must be implemented in performance critical application areas. If the overhead imposed by the transition system approach is unacceptable, unrestricted coding techniques combined with automated model extraction is recommended. The main advantage is the immediate feedback provided to programmers during implementation. However, this approach is only feasible if code for control flow is clearly separated from data manipulation code. As discussed in Chapter 4, this can be accomplished by implementing complex data structures as abstract data types. This allows detailed manipulation of fields in data messages to be encapsulated to support abstraction when deriving a model. Although the removal of such detail is important to ensure tractable models, it should be remembered that many defects may hide in such details. Testing is therefore essential to

check the correctness of the code not included in the generated models. Fortunately testing of such code is simplified by the structure imposed by using abstract data types. Because operations associated with each data structure are implemented as procedures, a small amount of overhead is unavoidable.

Future work

Software tools could be developed to support the transition system approach. For example, a program is needed to translate verified models automatically into the format of an executable transition system. Such a tool could be made to generate test cases to test all data manipulation code. In addition, execution of such tests could perhaps be partially automated. In practice, unstructured code is often encountered. Research to verify such code may help to strengthen the case for formal methods, but if such code could ever be verified, remains an unanswered question. Moreover, whether completely unstructured code should be tolerated in a society where computers are becoming more important in regulating our daily lives, is an open question.

Appendix A

A Transition system model

A model for the Alternating Bit protocol as a transition system:

```
/* ------------------------------------------------------- */
/*                  Macro Definitions                      */
/*                       Sender                            */
/*-------------------------------------------------------*/

/* Send data packet 0 */
#define SAction0() \
   Serial_Out ! 0; \
   event = 1

/* Receive acknowledgement 0 */
#define SAction1() \
   Serial_In ? data; \
   if \
   :: data == 0 -> \
      event = 2 \
   :: else -> \
      event = 6 \
   fi

/* Send data packet 1 */
```

```
#define SAction2() \
   Serial_Out ! 1; \
   event = 3

/* Receive acknowledgement 1 */
#define SAction3() \
   Serial_In ? data; \
   if \
   :: data == 1 -> \
      event = 0 \
   :: else -> \
      event = 6 \
   fi

/* ------------------------------------------------------- */
/*                   Macro Definitions                     */
/*                       Receiver                          */
/*-------------------------------------------------------*/

/* Receive data packet 0 */
#define RAction0() \
   if \
   :: Serial_In ? data -> \
      if \
      :: data == 0 -> \
         event = 1 \
      :: else -> \
         event = 6 \
      fi \
   :: timeout -> \
      event = 6 \
   fi

/* Compute checksum */
#define RAction1() \
   if \
```

```
   :: 1 -> \
      event = 2 \
   :: else -> \
      event = 6 \
   fi

/* Send acknowledgement 0 */
#define RAction2() \
   Serial_Out ! 0; \
   event = 3

/* Receive data packet 1 */
#define RAction3() \
   if \
   :: Serial_In ? data -> \
      if \
      :: data == 1 -> \
         event = 4 \
      :: else -> \
         event = 6 \
      fi \
   :: timeout -> \
      event = 6 \
   fi

/* Compute checksum */
#define RAction4() \
   if \
   :: 1 -> \
      event = 5 \
   :: else -> \
      event = 6 \
   fi

/* Send acknowledgement 1 */
#define RAction5() \
```

```
   Serial_Out ! 1; \
   event = 0

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

proctype InterSend(chan Serial_In, Serial_Out) {
   byte data;
   short event, state;

   state = 0;
   event = 0;
   do
   :: (state == 0) && (event == 0) ->
      state = 1;
      SAction0()
   :: (state == 0) && (event == 6) ->
      state = 3;
      SAction2()
   :: (state == 1) && (event == 1) ->
      state = 2;
      SAction1()
   :: (state == 2) && (event == 2) ->
      state = 3;
      SAction2()
   :: (state == 2) && (event == 6) ->
      state = 1;
      SAction0()
   :: (state == 3) && (event == 3) ->
      state = 0;
      SAction3()
   od
}

proctype InterReceive(chan Serial_In, Serial_Out) {
   byte data;
   short event, state;
```

```
    state = 0;
    event = 0;
    do
    :: (state == 0) && (event == 0) ->
       state = 1;
       RAction0()
    :: (state == 1) && (event == 1) ->
       state = 2;
       RAction1()
    :: (state == 1) && (event == 6) ->
       state = 0;
       RAction5()
    :: (state == 2) && (event == 2) ->
       state = 3;
       RAction2()
    :: (state == 2) && (event == 6) ->
       state = 0;
       RAction5()
    :: (state == 3) && (event == 3) ->
       state = 4;
       RAction3()
    :: (state == 4) && (event == 4) ->
       state = 5;
       RAction4()
    :: (state == 4) && (event == 6) ->
       state = 3;
       RAction2()
    :: (state == 5) && (event == 5) ->
       state = 0;
       RAction5()
    :: (state == 5) && (event == 6) ->
       state = 3;
       RAction2()
    od
}
```

```
proctype Serial_Link(chan Alt1In, Alt1Out, Alt2In, Alt2Out) {
   bit a;

   do
   :: Alt1In ? a ->
      if
      :: 1 ->
         skip
      :: 1 ->
         a = 1 - a;      /* corrupt the message */
      fi;
      Alt2Out ! a
   :: Alt2In ? a ->
      if
      :: 1 ->
         Alt1Out ! a
      :: 1 ->
         skip           /* Lose message */
      fi
   od
}

init {
   chan Alt1toSerial = [1] of { bit };
   chan Alt2toSerial = [1] of { bit };
   chan SerialtoAlt1 = [1] of { bit };
   chan SerialtoAlt2 = [1] of { bit };

   run Serial_Link(Alt1toSerial, SerialtoAlt1, Alt2toSerial, SerialtoAlt2);
   run InterSend(SerialtoAlt1, Alt1toSerial);
   run InterReceive(SerialtoAlt2, Alt2toSerial);
}
```

Appendix B

Alternating bit implemented in LF

```
MODULE AlternatingBit;

CONST
DataSize = 511;

TYPE
   DisplayPage = ARRAY 2000 OF RECORD
      char,attrib : UINT8
   END;

   IntType = RECORD
      x : UINT32;
      w : UINT16
   END;

   MsgType = RECORD
     Data : ARRAY Datasize OF UINT8;
     Bit : UINT8;
   END;

   MsgChan = [data(MsgType), ack(UINT8)];
   IntChan = [a(IntType)];
   TString = ARRAY 32 OF UINT8;
```

```
   TDispReq = [ s(TString) ];
   Array2 = ARRAY 12 OF UINT8;

PROCESS WriteString(  IN r : TDispReq  );
(*
   Basic display server
*)
VAR
   x,y,z,i,j : UINT32;
   s : TString;
   vp : DisplayPage AT  $0b8000;
   curPos : UINT16;
BEGIN
   i := 0; WHILE i < 2000 DO vp[i].char := 32; INC( i,1 ) END;
   x:= 0; y := 0;
   WHILE TRUE DO
      r ? s( s ) ;
      i := 0;
      WHILE (s[i] # 0 ) & (i < 32) DO
         IF (s[i] = 10) | (s[i] = 13 ) THEN (* CR or LF *)
             y := y+1; x := 0;
             IF y > 24 THEN y := 0 END
         ELSIF s[i] = 8 THEN (* backspace *)
            IF x > 0 THEN
               x := x-1;
                vp[ (y*80)+x ].char := 32
            END
         ELSIF s[i] = 9 THEN (* htab *)
            j := 0;
            WHILE j < 9 DO
               vp[ (y*80)+x ].char := 32;
               INC( j,1 ); INC( x,1 );
               IF x >= 79 THEN
                  x := 0; y := y+1;
                  IF y >= 24 THEN y := 0 END
               END
```

```
            END
         ELSE
            vp[ (y*80)+x ].char := s[ i ];   INC( x,1 )
         END;

         IF x >= 79 THEN
            x := 0; y := y+1;
           x := 0; y := y+1;
            IF y >= 24 THEN y := 0 END
         END;
         INC( i ,1 );          curPos := SHORT((y*80)+x);
         PORTOUT( $3d4,$e ); PORTOUT( $3d5,SHORT( curPos DIV $100 ));
         PORTOUT( $3d4,$f ); PORTOUT( $3d5,SHORT( curPos MOD  $100) )
      END
   END
END WriteString;

PROCESS WriteInt(IN IntC : IntChan; OUT scr : TDispReq);
VAR
   Int1 : IntType;
   i, w: UINT16;
   x: UINT32;
   x0 : UINT32;
   a: Array2;
   s:  TString;
BEGIN
   WHILE TRUE DO
      IntC ? a(Int1);
      x := Int1.x;
      w := Int1.w;
      x0 := x;
      i := 0;
      REPEAT
         a[i] := SHORT(SHORT((x0 MOD 10) + 48));
         x0 := x0 DIV 10;
         i := i + 1
```

```
      UNTIL x0 =  0;
      s[0] := 32;          (* " " *)
      s[1] := 0;
      WHILE w > i DO
         scr ! s( s  );
         DEC( w, 1 )
      END;
      s[0] := 45;          (* "-" *)
      IF x <  0 THEN
         scr ! s( s  )
      END;
      REPEAT
         DEC( i,1 );
         s[0] := a[i];
         scr ! s( s  )
      UNTIL i = 0
   END
END WriteInt;

PROCESS GenNewData(OUT datamsg : MsgChan);
VAR
   NewDataMsg : MsgType;
   Counter : INT32;
   Data    : UINT8;
BEGIN
   WHILE TRUE DO
      Counter := 1;
      Data := 0;
      WHILE Counter <= DataSize DO
         NewDataMsg.Data[Counter] := Data;
         Data := (Data + 1) % 10;
         Counter := Counter + 1
      END;
      datamsg ! data(NewDataMsg)
   END;
END GenNewMsg;
```

```
PROCESS Receiver(IN datamsg : MsgChan; OUT ackmsg : MsgChan;
                 OUT DispInt : IntChan; OUT DispStr : TDispReq);
VAR
   SenderMsg : MsgType;
   ControlBit : UINT8;
   s : TString;
   Int1 : IntType;
BEGIN
   ControlBit := 0;
   WHILE TRUE DO
      REPEAT
         datamsg ? data(SenderMsg);
         ackmsg ! ack(SenderMsg.Bit)
      UNTIL SenderMsg.Bit = ControlBit;
      ControlBit := 1 - ControlBit;
      (* Display received data *)
   END
END Receiver;

PROCESS Sender(IN ackmsg : MsgChan; OUT datamsg : MsgChan;
               IN gendatamsg : MsgChan);
VAR
   DataMsg : MsgType;
   AckMsg : UINT8;
   ControlBit : UINT8;
BEGIN
   ControlBit := 0;
   WHILE TRUE DO
      gendatamsg ? data(DataMsg);
      DataMsg.Bit := ControlBit;
      REPEAT
        datamsg ! data(DataMsg);
        ackmsg ? ack(AckMsg)
      UNTIL AckMsg = DataMsg.Bit;
      ControlBit := 1 - ControlBit
```

```
   END
END Sender;

PROCESS Init;
VAR
   ackmsg : MsgChan;
   datamsg : MsgChan;
   gendatamsg : MsgChan;
   IntC : IntChan;
   srq : TDispReq;
BEGIN
   NEW( srq );
   WriteString( srq );
   NEW( IntC );
   WriteInt( IntC, srq );
   NEW( datamsg );
   NEW( ackmsg );
   NEW( gendatamsg );
   GenNewData( gendatamsg );
   Sender( ackmsg, datamsg, gendatamsg );
   Receiver( datamsg, ackmsg)
END Init;

BEGIN
   Init
END AlternatingBit.
```

Appendix C

Alternating Bit Translated

A model for the Alternating Bit protocol translated with the L2P tool:

```
#define DataSize 2

typedef MsgType {
  byte Data[DataSize];
  byte Bit
}

proctype GenNewData(chan datamsg) {
  MsgType NewDataMsg;
  int Counter;
  byte Data;

  do
  :: 1 ->
    Counter = 1;
    Data = 0;
    do
    :: Counter <= DataSize ->
      NewDataMsg.Data = Counter;
      Data = (Data + 1) % 10;
      Counter = Counter + 1
```

```
    :: !(Counter <= 2) ->
      break;
    od;
    datamsg ! NewDataMsg
  :: !(1) ->
    break
  od
}

proctype Receiver(chan datamsg; chan ackmsg) {
  MsgType SenderMsg;
  byte ControlBit;
  bool MyRepeatTest1;

  ControlBit = 0;
  do
  :: 1 ->
    MyRepeatTest1 = 0;
    do
    :: (MyRepeatTest1 == 0) || !(SenderMsg.Bit == ControlBit) ->
      datamsg ? SenderMsg;
      ackmsg ! SenderMsg.Bit;
      MyRepeatTest1 = 1
    :: (MyRepeatTest1 == 1) && (SenderMsg.Bit == ControlBit) ->
      break
    od;
    ControlBit = 1 - ControlBit;
    printf("Data : %d\n", SenderMsg.Data)
  :: !(1) ->
    break
  od
}

proctype LossyChan(chan sndIn; chan sndOut; chan recvIn; chan recvOut){
  MsgType DataMsg;
  byte AckMsg;
```

```
  do
  :: sndIn ? DataMsg ->
    if
    :: recvOut ! DataMsg
    :: skip
    fi
  :: recvIn ? AckMsg ->
    if
    :: sndOut ! AckMsg
    :: skip
    fi
  od
}

proctype Sender(chan ackmsg; chan datamsg; chan gendatamsg) {
  MsgType DataMsg;
  byte AckMsg;
  byte ControlBit;
  bool MyRepeatTest2;

  ControlBit = 0;
  do
  :: 1 ->
    gendatamsg ? DataMsg;
    DataMsg.Bit = ControlBit;
    MyRepeatTest2 = 0;
    do
    :: (MyRepeatTest2 == 0) || !(AckMsg == DataMsg.Bit) ->
      datamsg ! DataMsg;
      do
      :: ackmsg ? AckMsg ->
        break
      :: timeout ->
        datamsg ! DataMsg
      od;
```

```
      MyRepeatTest2 = 1
    :: (MyRepeatTest2 == 1) && (AckMsg == DataMsg.Bit) ->
      break
    od;
    ControlBit = 1 - ControlBit
  :: !(1) ->
    break
  od
}

proctype Init() {
  chan recvIn = [0] of { byte };
  chan recvOut = [0] of { MsgType };
  chan sndOut = [0] of { byte };
  chan sndIn = [0] of { MsgType };
  chan gendatamsg = [0] of { MsgType };

  run GenNewData(gendatamsg);
  run Sender(sndOut, sndIn, gendatamsg);
  run Receiver(recvOut, recvIn);
  run LossyChan(sndIn,sndOut,recvIn,recvOut);
}

init {
  run Init()
}
```

Bibliography

[1] A. Arnold. *Finite Transition Systems.* International Series in Computer Science. Prentice Hall, 1994.

[2] T. Ball and S.K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *In Proc. of the 29th Annual ACM SIGPLAN-SIGACT Symposium on principles of Programming Languages*, pages 1–3, Portland, 16-18 June 2002.

[3] K.A. Bartlet, R.A. Scantlebury, and P.T. Wilkenson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. In *Communications of the ACM 12(5)*, pages 260–261, 1969.

[4] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification: model-checking techniques and tools.* Springer-Verlag, Berlin Heidelberg New York, 2001.

[5] P. Brinch Hansen. Joyce—A Programming Language for Distributed Systems. *Software—Practice and Experience*, 17(1):29–50, January 1987.

[6] T. Cattel. Modelization and Verification of a Multiprocessor Realtime OS Kernel. In *Proceedings 7th International Conference on Formal Description Techniques*, pages 35–50, Bern, Switzerland, October 1994.

[7] Toong Shoon Chan and Ian Gorton. Formal Validation of a High Performance Error Control Protocol Using SPIN. *Software-Practice and Experience*, 26(1):105–124, January 1996.

[8] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[9] Marina del Rey. Transmission Control Protocol, RFC0793, September 1981. (Available through electronic mail: mail to rfc-info@ISI.EDU).

[10] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[11] G. Duval and J. Julliand. Modeling and Verification of the RUBIS μ-Kernel with SPIN. In *In Proceedings of SPIN95, the First International Workshop on SPIN*, INRS-Télécommunications, Montréal, Quebec, Canada, October 1995.

[12] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Hongjun Zheng, and W Visser. Tool-Supported Program Abstraction for Finite-State Verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[13] Clarke Edmund M., Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, London, England, 2000.

[14] N. Francez. *Fairness.* Springer-Verlag, Inc., New York, 1986.

[15] Rob Gerth. Concise Promela Reference. Language reference, June 1997. (Available at http://spinroot.com/spin/Man/Quick.html).

[16] W. Geurts, K. Wijbrans, and J. Tretmans. Testing and formal methods — Bos project case study. In *EuroSTAR'98: 6^{th} European Int. Conference on Software Testing, Analysis & Review*, pages 215–229, Munich, Germany, November 30 – December 1 1998. Aimware, Mervue, Galway, Ireland.

[17] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *6-th IEEE Symposium on Logic in Computer Science*, pages 406–414, Amsterdam, 15-18 July 1991.

[18] Klaus Havelund and Jens Ulrik Skakkeæk. Applying Model Checking in Java Verification. *Lecture notes in computer science*, 1680:216–231, 1999.

[19] Gerard J. Holzmann. An Improved Reachability Analysis Technique. *Software Practice and Experience*, 18(2):137–161, February 1988.

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

[21] Gerard J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In R.Langerak, editor, *Proceedings of the 3th International SPIN Workshop*, Twente University, The Netherlands, April 1997.

[22] Gerard J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In R.Langerak, editor, *Proceedings of the 3th International SPIN Workshop*, Twente University, The Netherlands, April 1997.

[23] Gerard J. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[24] Gerard J. Holzmann. Designing Executable Abstractions. In *Proceedings of Formal Methods in Software Practice*, Clearwater Beach, Florida, USA, March 1998. ACM Press. Invited keynote address.

[25] Gerard J. Holzmann. Logic Verification of ANSI-C code with SPIN. In *SPIN model checking and software verification*, pages 131–147. Lecture Notes in Computer Science 1885, Springer Verlag, August 30 – September 1 2000.

[26] Gerard J. Holzmann. ON-THE-FLY, LTL MODEL CHECKING with SPIN, January 2003. (Available at http://spinroot.com/spin/whatispin.html).

[27] Gerard J. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proceedings: FORTE 1994*, pages 177–191, Berne, Switzerland, October 1994.

[28] Gerard J. Holzmann and Margaret H. Smith. Software Model Checking: Extracting Models from Source Code. In *FORTE/PSTV Conference*, Beijing, China, October 1999.

[29] Gerhard J. Holzmann. An Analysis of Bitstate Hashing. In *Proceedings of the 15th Symposium on Protocol Specification, Testing and Verification*, pages 301–314, London, 1995. Chapman & Hall.

[30] G.J. Holzmann. From Code to Models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, Newcastle upon Tyne, U.K., June 2001.

[31] G.J. Holzmann and M.H. Smith. Automating Software Feature Verification. *Bell Labs Technical Journal*, 5:72–87, April – June 2000.

[32] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model Checking for Managers. In *Theoretical and Practical Aspects of SPIN Model Checking*, pages 92–107. Lecture Notes in Computer Science 1680, Springer Verlag, July 5 – September 24 1999.

[33] Guoping Jia and Susanne Graf. Verification Experiments on the MASCARA Protocol. In *Proceedings of the 8th Spin Workshop*, Toronto, Canada, May 2001.

[34] J.J.D. Bull and P.J.A. de Villiers. Using SPIN to verify Protocols at the Implementation Level. In Paula Kotzé, John Barrow, and Lucas Venter, editors, *Proceedings of SAICSIT 2002*, pages 195–204, Port Elizabeth, South Africa, September 2002. ACM International Conference Proceedings Series.

[35] P.C. Jorgensen. *Software Testing: A Craftsman's Approach.* CRC Press, Inc., 1995.

[36] Phil Karn and Craig Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, 1991.

[37] Pim Kars. Experience using Spin and Promela in the Design of a Storm Surge Barrier Control System. In *In Proceedings of SPIN95, the First International Workshop on SPIN*, INRS-Télécommunications, Montréal, Quebec, Canada, October 1995.

[38] Pim Kars. The Application of Promela and Spin in the BOS Project. In *Proceedings of the 2nd SPIN Workshop*, Rutgers University, New Jersey, USA, August 1996.

[39] Paolo Maggi and Riccardo Sisto. Using SPIN to Verify Security Properties of Cryptographic Protocols. In *Proceedings of the 9th SPIN Workshop*, Grenoble, France, April 2002.

[40] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[41] Craig Partridge. *Gigabit Networking.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, Inc, One Jacob Way, Reading, Massachusetts 01867, 1994.

[42] Doron A. Peled. *Software Reliability Methods.* Texts in Computer Science. Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 USA, 2001.

[43] John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *22nd International Conference on Software Engineering*, pages 488–497, June 2000.

[44] T. Ruys and R. Langerak. Validation of Bosch' Mobile Communication Network Architecture with SPIN. In *In Proceedings of SPIN97, the Third International Workshop on SPIN*, University of Twente, Enschede, The Netherlands, April 1997. (Also available from URL:http://netlib.bell-labs.com/netlib/spin/ws97/ruys.ps.Z.).

[45] F. Schneider, Steve M. Easterbrook, John R. Callahan, and Gerard J. Holzmann. Validating Requirements for Fault Tolerant Systems using Model Checking. In *Proc. International Conference on Requirements Engineering, ICRE*, pages 4–14, Colorado Springs, Co., USA, April 1998.

[46] Siegfried Löffler and Ahmed Serhrouchni. Creating Implementations from Promela Models. In *Proceedings of the 2nd SPIN Workshop*, Rutgers University, New Jersey, USA, August 1996.

[47] Richard W. Stevens. *TCP/IP Illustrated: The Protocols*, volume 1 of *Addison-Wesley professional computing series.* Addison-Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, 1994.

[48] Richard W. Stevens. *TCP/IP Illustrated: The Implementation*, volume 2 of *Addison-Wesley professional computing series.* Addison-Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, 1998.

[49] Jan Tretmans, Klaas Wijbrans, and Michel R. V. Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System - Revisiting Seven Myths of Formal Methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[50] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets*, volume II, pages 1–22, Bonn, West Germany, 1989.

[51] F.A. van Riet. LF: A Language for Reliable Embedded Systems. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, November 2001.

[52] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - Second Generation of a Java Model Checker. In *In Proc. of Post-CAV Workshop on Advances in Verification*, Chicago, Illinois, July 2000.

[53] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *In Proc. of the 15th International Conference on Automated Software Engineering (ASE)*, Grenoble, France, September 2000.

[54] P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proceedings Computer-Aided Verification, LNCS 693*, pages 59–70, Elounda, Crete, June 1993. Springer-Verlag.

[55] Clement Yuen and Wei Tjioe. Modeling and Verifying a Price Model for Congestion Control in Computer Networks Using Promela/Spin. In *Proceedings of the 8th Spin Workshop*, Toronto, Canada, May 2001.