Automated program generation

Bridging the gap between model and implementation

Johannes A. Bezuidenhout

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

Supervised by: Dr. Jaco Geldenhuys

March 2007

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.

Signature: [illegible] Date: 05/03/2007

Abstract

The general goal of this thesis is the investigation of a technique that allows model checking to be directly integrated into the software development process, preserving the benefits of model checking while addressing some of its limitations. A technique was developed that allows a complete executable implementation to be generated from an enhanced model specification. This included the development of a program, the GENERATOR, that completely automates the generation process. In addition, it is illustrated how structuring the specification as a transitions system formally separates the control flow from the details of manipulating data. This simplifies the verification process which is focused on checking control flow in detail. By combining this structuring approach with automated implementation generation we ensure that the verified system behaviour is preserved in the actual implementation. An additional benefit is that data manipulation, which is generally not suited to model checking, is restricted to separate, independent code fragments that can be verified using verification techniques for sequential programs. These data manipulation code segments can also be optimised for the implementation without affecting the verification of the control structure. This technique was used to develop a reactive system, an FTP server, and this experiment illustrated that efficient code can be automatically generated while preserving the benefits of model checking.

Opsomming

Hierdie tesis ondersoek 'n tegniek wat modeltoetsing laat deel uitmaak van die sagteware-ontwikkelingsproses, en sodoende betroubaarheid verbeter terwyl sekere tekorkominge van die tradisionele modeltoetsing proses aangespreek word. Die tegniek wat ontwikkel is maak dit moontlik om 'n volledige uitvoerbare implementasie vanaf 'n gespesialiseerde model spesifikasie te genereer. Om die implementasie-generasie stap ten volle te outomatiseer is 'n program, die GENERATOR, ontwikkel. Daarby word dit ook gewys hoe die kontrolevloei op 'n formele manier geskei kan word van data-manipulasie deur gebruik te maak van 'n staatoorgangsstelsel struktureringsbenadering. Dit vereenvoudig die verifikasie proses, wat fokus op kontrolevloei. Deur dié struktureringsbenadering te kombineer met outomatiese implementasie-generasie, word verseker dat die geverifieerde stelsel se gedrag behou word in die finale implementasie. 'n Bykomende voordeel is dat data-manipulasie, wat gewoonlik nie geskik is vir modeltoetsing nie, beperk word tot aparte, onafhanklike kode segmente wat geverifieer kan word deur gebruik te maak van verifikasie tegnieke vir sekwensiëele programme. Hierdie data-manipulasie kode segmente kan ook geoptimeer word vir die implementasie sonder om die verifikasie van die kontrole struktuur te beïnvloed. Hierdie tegniek word gebruik om 'n reaktiewe stelsel, 'n FTP bediener, te ontwikkel, en dié eksperiment wys dat doeltreffende kode outomaties gegenereer kan word terwyl die voordele van modeltoetsing behou word.

Acknowledgements

I wish to express my sincere gratitude and appreciation to the following persons and institutions:

- My Heavenly Father.
- The Computer Science Department of the Stellenbosch University for their assistance.
- My promoter, Dr. Jaco Geldenhuys, for his guidance, encouragement, critical evaluation and assistance during this study.
- My parents, for their love, support and encouragement during all my years of studying.
- My wife, Mareli, for being her exceptional self, for her constant love and support, and all the cups of coffee.

Contents

Chapter 1

Introduction

Formal methods aim to bring software engineering in line with the more traditional engineering disciplines. Because of the need to balance the issues of cost and reliability, the use of these methods are usually only justified when the cost of possible errors is very high, as is the case with applications involving safety and security. As formal methods evolve and mature, and growing computational power diminish previous practical limitations, some of the focus is shifting from achieving basic functionality to enhancing formal methods in terms of effectiveness and usability. One example of this is the relatively new trend of *software model checking* [54], where the focus has shifted from verifying hand-built models to applying model checking directly at the implementation level. One way to achieve this is to automatically generate an executable implementation from the verified specification, and this thesis investigates such an approach.

Model checking is an automated formal method for the verification of finite-state concurrent systems. Due to the nature of concurrent systems, normal development tools are insufficient in general. Just reasoning about the expected behaviour of concurrent systems without the use of automated tools is extremely hard, even impossible. This is because of the possibly infinite execution interleavings. An erroneous execution of a concurrent system depends on a specific combination of the execution paths of each of the processes in the system, which in turn depends on the timing of events. Since this exact execution path is only one of a possibly infinite number of execution paths of the system, testing techniques may not reveal such behaviour, and is therefore not a satisfactory method for guaranteeing correctness or reliability. Even if an erroneous execution were detected it would, in general, be impossible to reproduce, since even with the same input variables it is not possible to control the exact timing of all events. Model checking addresses these issues by allowing the verification of a requirements property for all possible executions of a given system specification, in a formal and highly automated manner.

The traditional model checking approach is to manually construct an abstract model of the system, which is is then verified against its formal requirements specification. A model checker checks every possible execution of the model and verifies that none of the paths in the reachable state space violates the formal specification. If a violation is detected, a counterexample is generated in the form of an execution trace that leads to the violation. Models are described in a special modelling language which is usually specific to the model checker, and is also generally more restrictive than a general programming language, so as to facilitate analysis and verification.

The fact that the traditional model checking approach is based on a manually constructed model of the system is one of its biggest limitations — the keywords being **model** and **manual**. The problem is that there exists a discrepancy, or gap, between the specification that is verified (the model) and the specification that represents the actual implementation (the implementation source code). This is what we refer to as the *specification gap*, and it is the source of two major problems. Firstly, a verified model does not guarantee that the implementation satisfies the requirements if informal implementation techniques are used. Even with the use of formal methods, if there are any manual steps involved in the translation between model and implementation, the possibility always exists that errors are introduced that will invalidate the verification. Secondly, manually constructing a model that is both accurate as well as tractable is a non-trivial exercise, introducing issues of *timing* and *relevance*. If the effort of constructing the abstract model is not significantly less than the actual development effort itself, the verification effort will either hold up the development process, or quickly start to lag behind the current version of the system and eventually become incomplete, inaccurate, or even completely irrelevant. These are some of the main reasons that led to research in the formal methods community to be directed at development of techniques for applying model checking directly at the implementation level, rather than on system model specifications.

It is currently possible to distinguish between three different general approaches in software model checking, based on how the program source is interpreted: *model extraction*, *program generation* and *model checking programming languages directly*. Model extraction refers to automatically generating abstract models from existing implementations, program generation refers to automatically generating correct implementation skeletons from verified models, and directly model checking programs means that entire implementations are used as input to the model checker.

This thesis investigates a technique where a full implementation is generated from a verified model. To formalise the translation step we propose using the same structure (a state transition system) for the model and the implementation. A transition system approach allows us to separate the flow of control from data manipulation. This benefits verification as well as testing. Model checking focuses on verifying the correct flow of control, and testing now has only to be applied to clearly designated pieces of code that perform data manipulation. These code fragments can be tested independently and in isolation from the environment of the system. This structuring approach has been shown to be effective in [15] where implementations are derived from verified formal specifications. Whereas the authors of [15] manually derive the implementations, we will automatically generate a full implementation from a verified model. The design and implementation of a generator tool that can parse specially enhanced models and generate full implementations forms the core of this thesis.

It was decided to use the SPIN verification system as a framework for our tool since it is mature, well-known, and freely available. The input language for SPIN, PROMELA, is also one of the more programming-language-like modelling languages. ANSI C was chosen as implementation language, for two reasons. Firstly, it is a popular programming language and one of our aims is to make model checking more attractive to the general programmer. Secondly, it integrates well into the SPIN system — PROMELA resembles C and SPIN itself is implemented in C. Since concurrent reactive systems [49], such as communication protocols, are well suited to the transition system approach, we restrict our focus to this type of application.

The thesis is structured as follows. Chapter 2 is a survey of the relevant background literature and covers model checking, the SPIN verification system, and software model checking. Chapter 3 explores the approach taken in this investigation and documents the design and implementation of the GENERATOR, the program that automates the implementation generation step. Chapter 4 contains case studies that demonstrate the application of the proposed technique and the use of the GENERATOR. Conclusions are presented in Chapter 5.

Chapter 2

A Survey of Software Verification

This chapter provides a survey of software verification in as much as it pertains to the use of model checking in the software development process. This will provide the necessary background for the tool we introduce in the next chapter. Much of the information here comes from the summary articles of Clarke [19, 23], Holzmann [61], and Katoen [80].

The goal of *software verification* is to ensure that a system conforms to its *requirements specification*, where it is assumed that the requirements correctly represent what is really expected from the application. Verification techniques can be classified as either *static* or *dynamic* verification. Dynamic verification is performed by executing a system implementation, and includes testing and simulation, or experimentation. Static verification, on the other hand, is a process of checking the description of a system against certain requirements by way of physical inspection. One example of static verification is *formal verification*.

2.1 Formal Verification

As Dijkstra pointed out, testing can only show the presence of errors, never their absence [31]. Formal verification on the other hand can prove, or disprove the correctness of a system description with respect to certain requirements. The basic idea behind this approach is as follows. A formal *model* is constructed which represents the possible behaviour of the system, and the correctness requirements for the system are written in a formal *requirements specification* that represents the desired behaviour of the system. Using formal methods it is then verified whether the possible behaviour corresponds to the desired behaviour.

Formal verification can be applied to *sequential* programs in the following fashion. First the desired behaviour is formalised using pre- and postconditions formulated in predicate logic. A *precondition* describes the set of initial states, and a *postcondition* describes the set of desired final states. Once this is done, the sequential program is coded in some abstract pseudo-code language, such as Dijkstra's guarded command language [32], and it is proven in a stepwise manner that the abstract system description satisfies the specification. The actual proof is constructed from a set of proof rules, which generally corresponds to program constructs. These rules are written in the form:

$$\{\phi\}S\{\psi\}$$

where ϕ is a precondition, S is a program statement, and ψ is a postcondition. This is known as a *Hoare triple*, due to the work of Tony Hoare [57]. This formula can be read as follows: Whenever ϕ holds in the state before the execution of S, then ψ will hold afterwards. The formula is partially correct if any terminating computations of S that starts in a state satisfying ϕ, terminates in a state satisfying ψ. It is totally correct if any computation of S that starts in a state satisfying ϕ, terminates and finishes in a state satisfying ψ.

Deterministic programs always provide the same results when provided a given input, and it naturally follows that this type of program lends itself more freely to the approach just described. The introduction of parallelism, however, also leads to the introduction of *non-determinism*. Concurrent processes can potentially interact at any point during their execution, which means that the results of a given execution may depend strongly on the order of execution. Also, very importantly, the execution of parallel systems does not necessarily terminate; this is the case for most reactive systems. Thinking of a program as a function that produces a certain desired output when provided certain allowed input is no longer appropriate when reasoning about concurrent systems in general. Instead of reasoning only in terms of the state of a system before and after an execution, we need to reason about the different executions of the system by making statements about the executions themselves.

Although various efforts have been made to generalise the classical formal verification approach as described above, it just does not scale well for realistic concurrent systems. The proof rules are very complex and the proof systems quickly become very large. This makes the process lengthy, tedious and easily susceptible to errors. *Proof assistants* and *theorem provers* are both tools that address some of these issues, but there is a formal method generally more suited to the verification of concurrent systems, namely *model checking*.

2.2 Model Checking

Model checking is an automated, model-based verification technique for finite concurrent systems. Given a description of a system in the form of a finite-state model (the possible behaviour) and a description of the requirements specification (the desired behaviour) for that system, a model checker automatically verifies whether the modelled system satisfies the requirements. If the model satisfies the requirements, the model checker returns a positive result, and negative otherwise. In the case of a negative result, the model checker can (in most cases) provide a counterexample that indicates exactly how the requirements are violated. This is usually in the form of an execution trace of the system model. This verification methodology is illustrated in Figure 1.

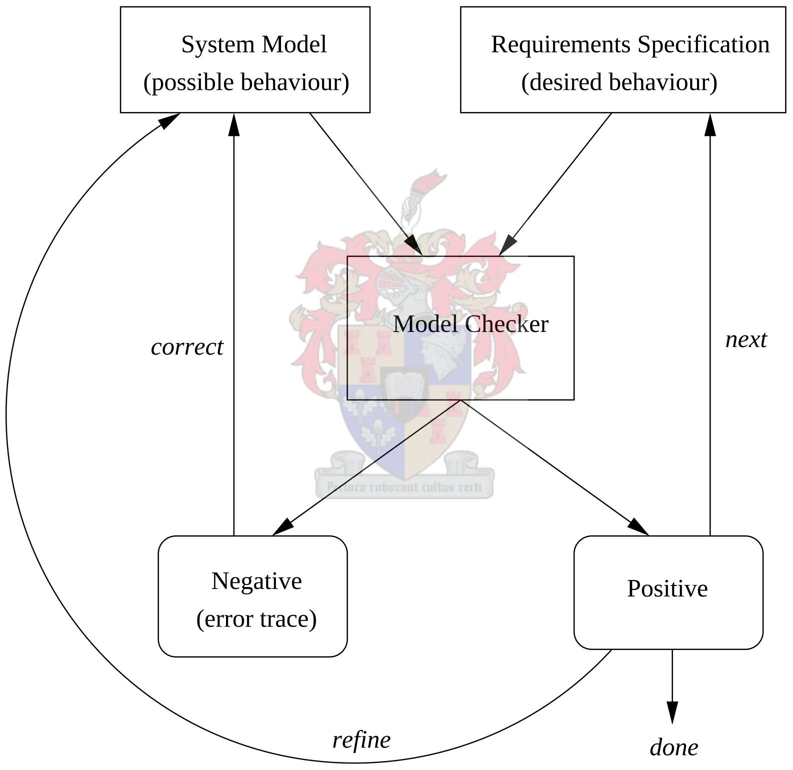

Figure 1: Model Checking Methodology.

The system description or model is an abstract representation of the system, and should only represent the behaviour of the system necessary to verify the specific requirements. The reason for this is that a model checker can in general not efficiently handle the complexity of a complete system implementation. Moreover, excluding irrelevant details may make the model more comprehensible for users of the verification, and therefore produce more useful counterexamples.

There are two basic approaches to applying this verification methodology to the software development process. The first, and also most ideal, approach is to develop a correct, i.e., verified, system description or model of the system, and then derive the actual implementation from the verified model. The second approach is to derive a system model from a preexisting implementation. The reason the first approach is considered ideal is that the earlier problems are identified in the development of an application, the less costly they are to rectify. However, the second approach is unfortunately more representative of real-world development practices. This is also the only option if verification has to be applied to a previously developed software application.

Regardless of whether the implementation is derived from the model, or vice versa, the verification results obtained for the model is ultimately translated to the system implementation. If the verification is successful, the model can be further refined and the verification repeated, until the model is considered sufficiently representative of the system. It is also possible that the requirements were specified incorrectly, in which case the result of the verification is called a *false negative*. If the model was derived from an actual implementation, the counterexample produced can be checked against the implementation to determine its validity. If the behaviour can be reproduced in the implementation, it represents a true error, and the implementation can be corrected. If the behaviour is not reproducible, it indicates that the system model is not sufficiently accurate, and can be used to refine the model.

The basic operation of a typical model checker can be described as an *exhaustive state space search* of the system model. This means that the model checker checks that for each state of the model, the desired behaviour is satisfied. In its most basic form this is known as *reachability analysis*. With reachability analysis it is possible to prove *invariant properties*, which are properties that should hold in every state of a computation, as well as *freedom from deadlock*. To check for deadlock it is sufficient to determine whether there exists a reachable state in the model without any successors. Simple reachability analysis is, however, insufficient for reasoning about the behaviour of the system over time, such as claims concerning the relative order of events.

2.2.1 Different Approaches to Model Checking

We can distinguish between two main approaches to model checking, based on the way the desired behaviour, or requirements specification, is represented. The first is a *logic-based* or *heterogeneous* approach, and the second a *behaviour-based* or *homogeneous* approach.

In the logic-based approach, the desired behaviour is captured as a set of properties in some appropriate logic. The system description, or possible behaviour, is usually modelled as some form of finite-state automaton. The system is considered correct with respect to the requirements if it satisfies the properties for a given set of initial states. Formally, this can be stated as

$$\mathcal{M}, S \models \phi$$

where $\mathcal{M}$ is the system, S is the set of initial states, and ϕ is a logic formula representing a requirements property. In formal logic, $\mathcal{M}, S$ is known as *model* for property ϕ, which is where the term *model checking* comes from.(We also refer to $\mathcal{M}$ as a model of the implementation, which may be slightly confusing, but generally does not lead to any problems.) This approach originates from the independent work of Clarke and Emerson [20, 21], and Queille and Sifakis [101]. Both these approaches are based on *temporal logic*, which was first introduced to Computer Science by Pnueli [98].

In the behaviour-based approach, both the desired and possible behaviours are specified using the same formalism, and equivalence relations (or pre-orders) are then used as a correctness criterion. Equivalence relations capture the notion "has exactly the same behaviour as", whereas pre-order relations represent the notion "has at least the same behaviour as". A system is then considered correct if the desired and the possible behaviour are equivalent (or ordered) with respect to the equivalence (or pre-order) attribute under investigation. This approach is most commonly associated with *process algebras*, pioneered by Milner [87, 88] and Hoare [58, 59].

From here on we will only be focusing on the logic-based model checking approach, with the requirements specified in temporal logic. In the literature this is sometimes referred to as *temporal logic model checking*.

2.2.2 Temporal Logic

As already mentioned, predicate or propositional logic is not adequate for reasoning about concurrent systems. We therefore turn to *temporal logic*, which allows us to express properties involving the relative ordering of events, without having to explicitly mention the time at which the events occur. The concept of temporal logic was already introduced in the 1960s by Arthur Prior, in the field of Philosophy, but its introduction to Computer Science for the purpose of specification and verification is due to Pnueli [98].

For example, if we were reasoning about the behaviour of a railway crossing controller, we would like to make statements like

A: When a train is in the crossing, the gate will always be down.

and

B: If the gate is down, it will eventually again be up.

Temporal logic extends propositional logic by introducing operators that express properties of system states over time. Examples of such operators are **U** (until), **G** (globally, or always), and **F** (future, or eventually). ϕ **U** ψ specifies that property ϕ holds in the current state and in all subsequent states until a state is reached where ψ holds, **G** ϕ specifies that ϕ holds in the current state and all subsequent states (in other words, it will always hold), and **F** ϕ specifies that ϕ holds in the current state or will hold in some future state. Using these operators we can formally define our statements as follows. We first define the atomic or primitive propositions:

$p \equiv$ "a train is in the crossing"
$g \equiv$ "the gate is down"

The first statement can then be expressed in temporal logic as

$$\mathbf{G}\ (p \Rightarrow g)$$

which reads: "always, if p then g", or by expanding the propositions: "it is always so that if a train is in the crossing, the gate is down". The second statement is a bit more tricky, and can be expressed as

$$\mathbf{G}\ (g \Rightarrow \mathbf{F}\ (\neg g))$$

where $\neg g$ indicates the negative form of g ("the gate is up"). This reads: "always, if g then eventually $\neg g$", or "it is always so that if the gate is down now, then in some future state the gate is up again".

Temporal logics come in two varieties:

1. *Linear time temporal logic* allows reasoning about an individual system execution as a linear sequence of events. The qualitative notion of time is linear, meaning that at each

moment of time there is only one possible successor state, and thus only one possible future. The best-known example of this kind is *propositional linear time logic*, or LTL, introduced by Pnueli [98].

2. *Branching time temporal logic* introduces the idea that at each moment there may be several different futures. In other words, for every state of the model there possibly exists multiple successor states. A statement in branching time temporal logic reasons about a *tree* of states, rather than a sequence, with each path in the tree representing a single possible computation. There exists various types of branching time temporal logic that differ in expressiveness. The best known example is *computation tree logic*, or CTL, introduced by Clarke and Emerson [20].

Although it may appear from the above description that linear time temporal logics are not suited for reasoning about concurrent systems, this is not so. For a system to satisfy a linear time property ϕ, all the executions of the system must satisfy the property.

These two varieties of temporal logic has led to two different directions of research in model checking. Both these directions have led to unique algorithms and implementations, and both have their own strengths and weaknesses. There are various verification tools based on each of these two approaches, some purely for research, but others that have been used to verify production software. Two examples of such tools are the SPIN [61] model checker for LTL, and the SMV [85] model checker for CTL.

For this thesis, it was decided to implement our technique within the SPIN verification system. There are several reasons for this decision, which will be pointed out later. At this point we can point out that SPIN is one of the most mature model checking tools. This claim is supported by the fact that SPIN has been successfully used for the verification of substantial and diverse applications [39, 52, 68, 74, 79, 102], as well as the SPIN workshop held annually since 1995. Moreover, the current and up-to-date SPIN source code is available freely online (`http://spinroot.com/`), and is well documented.

SPIN and SMV are only two examples of formal verification tools. Yahoda (`http://anna.fi.muni.cz/yahoda/`), an online database of verification tools, currently lists 43 model checking tools.

2.2.3 LTL Model Checking

The basic model checking problem is to verify that a formally defined system satisfies a formally defined requirements specification. The general approach for model checking LTL that we describe below is known as *automata-theoretic* model checking, and was first proposed by Vardi and Wolper [107, 113].

A *labelled finite state automaton*, or LFSA, is a tuple $(\Sigma, S, S^0, \rho, F, l)$, where:

- Σ is a non-empty set of symbols,
- S is a finite, non-empty set of states,
- $S^0 \subseteq S$ is a non-empty set of initial states,
- $\rho : S \rightarrow 2^S$ is a transition function,
- $F \subseteq S$ is a set of accepting states, and
- $l : S \rightarrow \Sigma$ is a labelling function for states.

We denote the set of finite sequences over Σ as Σ^*, and the set of infinite sequences as Σ^ω.

A *run*, σ, is a finite or infinite sequence of states, starting from an initial state such that each state in the sequence can be reached via a transition from its predecessor state. A run is *accepting* if it ends in an accepting state. A *finite word* $w = a_0 a_1 \ldots a_n$ is *accepted* by LFSA A if and only if there exists a finite run $\sigma = s_0 s_1 \ldots s_n$ such that $l(s_i) = a_i$ for $0 \leq i \leq n$ and $s_n \in F$. The *language* accepted by A, denoted $\mathcal{L}(A)$, is the set of finite words accepted by A:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid w \text{ is accepted by } A\}.$$

Since we are interested in proving properties that refer to infinite behaviour, we need to define how infinite runs are accepted. This is called *Büchi-acceptance*, and automata with this alternative definition of accepting runs are called Büchi or ω-automata. A labelled Büchi automaton, or LBA, can then be defined as an LFSA that accepts *infinite* words. An *infinite word* $w = a_0 a_1 a_2 \ldots$ is *Büchi-accepted* by LBA A if and only if there exists an infinite run $\sigma = s_0 s_1 s_2 \ldots$ such that $l(s_i) = a_i$ for $i \geq 0$ and for each i there exists $j \geq i$ such that $s_j \in F$. The language of an LBA A, denoted $\mathcal{L}_\omega(A)$, is then defined as the set of infinite words accepted by A:

$$\mathcal{L}_\omega(A) = \{w \in \Sigma^\omega \mid w \text{ is accepted by } A\}.$$

With this background we can now describe the basic scheme for LTL model checking. An overview of this process is shown in Figure 2.

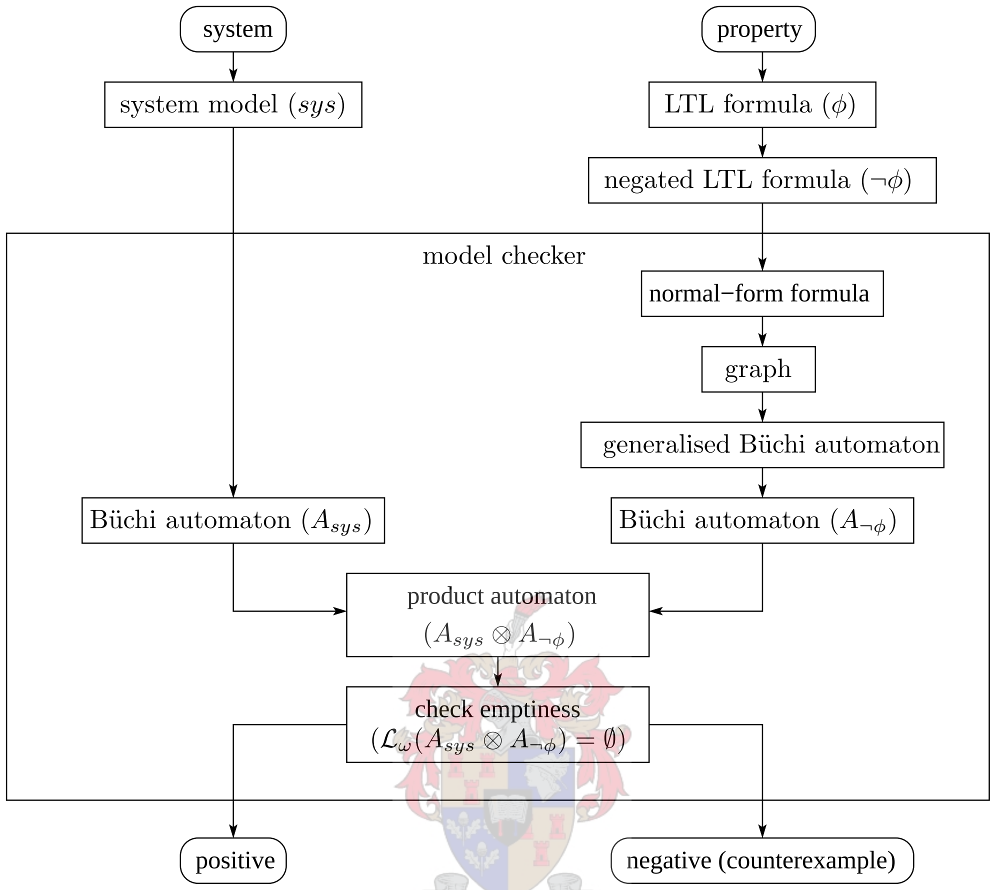
system
system model (sys)
property
LTL formula (ϕ)
negated LTL formula ($\neg\phi$)
model checker
normal−form formula
graph
generalised Büchi automaton
Büchi automaton (A_{sys})
Büchi automaton ($A_{\neg\phi}$)
product automaton ($A_{sys} \otimes A_{\neg\phi}$)
check emptiness ($\mathcal{L}_\omega(A_{sys} \otimes A_{\neg\phi}) = \emptyset$)
positive
negative (counterexample)

Figure 2: Model checking LTL.

The very first step is of course to formally define the system behaviour as well as the desired behaviour. The language used to model the system is specific to the model checker implementation, which means that the translation of the model to automata form is also model checker dependent. The desired behaviour is formalised as a requirements specification, with the properties specified as LTL formulas.

In the next step, the system model and requirement specification are translated to Büchi automata A_{sys} and A_ϕ, respectively. Vardi and Wolper showed that all LTL formulas can be translated to Büchi automata [108, 113], but most practical implementations are based on the more efficient algorithm in [38]. Thus, the language of the LBA A_{sys} describes the possible behaviour of the system model, and the language of the LBA A_ϕ describes the desired behaviour. We need to check that:

$$\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi)$$

in which case the system satisfies ϕ. Unfortunately, deciding language inclusion for Büchi automata is PSPACE-complete. This means that, according to our current knowledge, the problem requires a running that is at least exponential in the size of the two automata. Fortunately there is an alternative approach we can take. Computing whether the desired behaviour includes the possible behaviour, is equivalent to computing whether the undesired behaviour does not include any possible behaviour. This amounts to checking whether the runs accepted by A_{sys} and $\overline{A_\phi}$ are disjoint, or formally:

$$\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi) \Leftrightarrow (\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(\overline{A_\phi}) = \emptyset)$$

where $\overline{A}$ is the complement of A, and accepts the language $\Sigma^\omega \setminus \mathcal{L}_\omega(A)$. Complementation of Büchi automata is also PSPACE-complete, but again there is a workaround. The complement automaton of A_ϕ is equal to the automaton for the negation of ϕ:

$$\mathcal{L}_\omega(\overline{A_\phi}) = \mathcal{L}_\omega(A_{\neg\phi})$$

and we can therefore implement the model checking process for the negation of the formula, as Figure 2 illustrates. If the LTL formula ϕ represents a desired property, its negated form $\neg\phi$ represents an undesired property. The added negation does not significantly increase the complexity of the transformation. The Büchi automaton $A_{\neg\phi}$ is constructed by first translating the formula to normal form, then to a graph form, and then to a generalised Büchi automaton. (The details of these intermediate forms fall outside the scope of this thesis). The automaton represents all undesired behaviour. Since the size of the automaton can be exponential on the length of the formula, the automaton for the negated formula can possibly be larger than that for the original formula. The product of the automata for the system and the negated property ($A_{sys} \otimes A_{\neg\phi}$) represents all possible computations of the system that violate ϕ. Verifying whether the system model *sys* satisfies the property ϕ then boils down to checking the emptiness of the language accepted by the resulting automaton, which can be formally stated as the *emptiness problem*:

$$\mathcal{L}(A_{sys} \otimes A_{\neg\phi}) = \emptyset.$$

The language of the product automaton is empty if it does not accept any infinite words. If this is true, the system model does not contain any behaviour that violates property ϕ. If the language is not empty, it consists of the set of Büchi-acceptance runs that correspond to the executions that violate ϕ, and can then be used to produce counterexamples.

One of the advantages of this approach is that it allows for *on-the-fly* model checking. It is possible to check for violating accepting runs during construction of the product automaton ($A_{sys} \otimes A_{\neg\phi}$), and once a violation is detected, the model checker can immediately report it and abandon the construction. If the system satisfies the requirements, the complete product automaton has to be constructed. Fortunately, approaches such as abstraction techniques [23, 27], symmetry methods [22, 28, 34, 77], and partial order reduction [42, 69, 70, 95, 106] can also be applied to limit the size of the product automaton.

Examples of techniques that reduce the memory requirements of the model checker, as listed by Geldenhuys in [37], are state compaction [36] and byte masking, run-length encoding and Huffman encoding [67], closed (rehashing) hash tables, open (chaining) hash tables [41], bit-state hashing [60], hash compaction [103, 112], recursive indexing [62, 109], minimising automata-based techniques [71], sharing trees (or GETSs) [46], and difference compression [94].

2.2.4 The Spin Model Checker

Spin is an efficient verification system that supports the design and verification of asynchronous systems [61, 66]. Besides implementing LTL model checking, Spin also provides for the simulation of the system model specification. A graphical overview of the structure of the system is given in Figure 3. Spin is implemented as a command-line program, but a graphical front-end, XSpin, is also provided with the system. The direct command-line approach allows more flexibility and has the possibility of delivering a more specialised verification in the hands of an experienced user, whereas the graphical interface provides a more user-friendly approach by automatically generating many of the required options for the user, and providing menu options for the rest. XSpin also provides a graphical display of the system execution as a message sequence chart, which is often more helpful than the textual output provided by the command-line.

The system model specifications accepted by Spin are written in the language Promela, which is an acronym for PROcess MEta LAnguage. Promela is described in detail in the next chapter, so only a basic introduction is given here.

A Promela model specifies an asynchronous system of concurrently executing processes, that can communicate via message channels and shared variables. The language is heavily

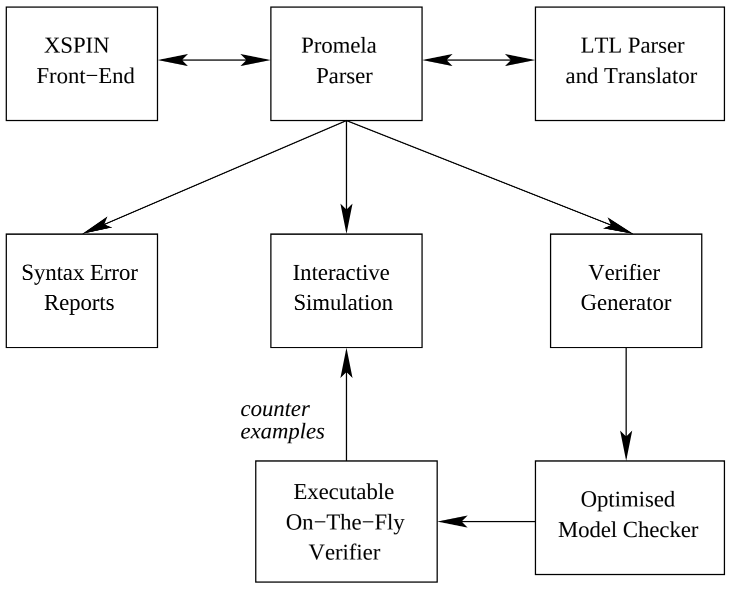

Figure 3: The structure of Spin simulation and verification.

influenced by the *guarded command language* of Dijkstra [32]. The syntax of expressions, declarations and assignments in Promela resemble that of the C programming language.

Typically, the first step is to use Spin to perform a random simulation of the system. This is an easy way to check that the model at least appears to behave as expected. The next step is to actually prove that it is behaving correctly in terms of the requirements. This is done by expressing the formal requirements as LTL formulas and using Spin to model check the system specification against the correctness requirements.

Spin generates an optimised on-the-fly verifier as an ANSI C program source based on the system and requirement specifications. This program performs the actual verification and implements the LTL model checking approach described in the previous Section, albeit with many more options and features. The program is first compiled and then executed. During all three of these steps, generation, compilation and execution of the C program, it is possible to specify additional options to fine tune the verification. For example, it is possible to add options when compiling the program to specify which reduction algorithms are to be used. When executed, the verifier returns with a positive result if there are no property violations, and provides a counterexample in the case of a violation. The counterexample is in the form of an execution trail of the model, and the simulation functionality of Spin can be used to perform an interactive playback of the trail to examine the cause of the violation.

By default, Spin checks for the absence of deadlock, unspecified reception of messages and unexecutable code. Additionally it is also possible to verify the model against specific correctness requirements by using inline assertions, state labels, LTL properties or never-claims. The PROMELA assert statement works exactly like its programming language counterpart, and provides an easy way to check simple safety properties. The statement takes any valid PROMELA expression as argument and traps violations of the property specified by the expression. This works in simulation as well as verification mode, but the simulation of course only checks the assertion for one random execution, whereas the verification checks it for all possible executions of the system. Another way to make specific assertions about the model behaviour is through the use of state labels. End-state labels are used to indicate valid end states, that should not be seen by SPIN as constituting a deadlock. In the same way it is also possible to define progress-state labels. These are used to specify liveness properties, and marks a state that must be visited infinitely often in any infinite system execution. Another label for specifying liveness properties is the acceptance-state label. These are used to formalise Büchi acceptance conditions, but since they are in most cases automatically generated from LTL formulas, it is rarely necessary to manually add them. To verify anything more than basic correctness properties, it is necessary to specify the desired behaviour as a never-claim or LTL formula.

A PROMELA never-claim is a formalism that describes the desired behaviour of the system, in the same way that the model specifies the possible behaviour. Never-claims are more expressive that LTL, and any LTL formula can be translated to a corresponding never-claim. Although never-claims are more expressive, LTL formulas are in general easier to formulate. Recent versions of SPIN therefore allow requirements to be specified as LTL formulas, and automatically derive the corresponding never-claims. This process uses the algorithm from [38] with some optimisations from [35]. SPIN does not allow the use of the **X** (next) temporal operator, so the logic accepted by SPIN is referred to as LTL-X. One other subtle difference is that SPIN never-claims also accept finite runs. This is achieved by applying a *stutter extension* to a finite run to make sure it is Büchi-accepted. A stutter extension of a finite run σ of finite state automaton A is the infinite run that is derived from σ by appending an infinite number of copies of the last state.

This was just a general introduction to the SPIN system. For a complete description of the tool the reader is referred to the SPIN reference book [65]. The SPIN website `http://spinroot.com/` is another source, and contains tool documentation, tutorials, sample applications, the actual program source, as well as information about SPIN workshops.

2.3 Models versus Implementations

It is only recently that the development of efficient algorithms and ever growing computing power allowed model checking to become a practically viable option for formal verification method. Active research in the field has produced steady improvement in the efficiency and usability of the technology, and computing power per unit cost has continued to evolve according to *Moore's Law* [89]. This has led to model checking being successfully applied to the development of reliable software as well as debugging of existing systems. In recent years these advances have led to a new direction in model checking research, where the attention has shifted from the development of the basic capability to perform logic model checking at the design level on abstract system models, to the development of methods that allow the application of logic model checking directly at the implementation level. Before discussing these methods, let us first take a look at the benefits and limitations of the basic, or traditional, model checking approach that led to this development. In describing model checking we have already touched on some of these.

2.3.1 Benefits of Model Checking

Model checking is a formal method based on a sound mathematical foundation. This makes it appealing from a Computer Science perspective, since we can scientifically reason about it. Software engineering on the other hand, as an engineering discipline, focuses on the application of the technique to software development.

Model checking is **highly automated**. Given a finite-state system description and the LTL formulation of the correctness requirements, a model checker can automatically check whether the property holds. Other techniques (such as automated theorem proving) requires significantly more expertise and effort to achieve the same result.

A verification run returns a **definite 'yes' or 'no' answer** (if the question is at all decidable). In the case of a property violation, a **counterexample** is produced to show the exact behaviour that causes the violation.

Erroneous behaviour is **reproducible**. Concurrent behaviour of reactive systems is highly dependent on intricate event timings, which makes it difficult to reproduce errors with testing. Model checking systematically examines all possible behaviours, so that, if a property violation is matched once, any future verifications will match the exact same violation (for the same model and property). Also, the counterexample can then be used to trace the precise steps in the model execution that led up to the error.

Model checking supports **partial verification**, which means that it can be applied to a partial design. Only the behaviour relevant to the requirement has to be modelled, and it is possible to tailor the specification to the specific verification. This can result in improved efficiency.

Model checking is a **general approach** not fixed to any one specific application area. This includes hardware as well as software, high level designs as well as low level embedded systems, and even business models [78].

When applied in the context of formal software development, model checking does not necessarily cost more than testing and simulation, and has been shown to be more effective in its intended application area. In some cases the use of model checking has even been shown to reduce development time.

2.3.2 Limitations of Model Checking

Model checking is subject to **decidability issues**. Generally speaking, even simple properties can be formally undecidable. Model checking is based on the exploration of the system state space, which means that its application is restricted to systems with behaviour that can be modelled as **finite-state automata**. Event-driven, distributed, and concurrent systems, which we are focusing on, all fall in this category.

Given a finite-state model, another fundamental issue for model checking is the **state space explosion problem**. For a model checker to effectively traverse the state space, the states has to fit in primary memory. However, the size of a system's state space is in the worst case proportional to the Cartesian product of the state spaces of its components. The state representation can be swapped in and out of primary memory as needed, but this is inefficient. The verification is therefore limited by the physical constraints of memory and time. It is possible to only check a partial state space, but then it cannot be shown that the property being checked holds for the complete system behaviour. Any errors found in the explored state space are, however, valid errors. Effective algorithms and techniques, as well as the constant increase in available computing resources has ameliorated the impact of this problem.

Due to the nature of model checking it is more suited to the verification of *flow of control* than *data manipulation* in a system. Data intensive operations usually do not carry a lot of useful state information when compared to the significant increase in state space caused by the large data ranges.

A model checker implementation, as a software application, might itself be unreliable. However, since model checking is based on standard and well known logically sound algorithms, confidence is achieved in model checker implementations through thorough verification and intensive use. This is similar to the situation for compilers. In some cases theorem provers have been used to verify components of the model checker implementation.

In contrast to theorem proving, it is in general not possible to check generalisations with model checking. Properties verified for a fixed number of variations does not scale to the general case. Using model checking to prove a property for specific cases, it could, however, be possible to use an automated theorem prover to generate the proof for the general case.

This brings us to the issue that we aim to address in this thesis, the *model construction problem*. The problem is twofold, in that the process of constructing a system model can be both **error-prone** and **time consuming**. (This argument is of course equally valid for the process of deriving the implementation from the model.) The problem stems from the fact that there exists a discrepancy, or gap, between the artifacts representing the actual system implementation and those accepted by most current verification tools. This is due to the difference in semantics, as well as syntax, of the specification formats used for implementation and modelling. Software is developed using general-purpose programming languages, such as C and Java, while most verification tools accept specification languages designed to be more convenient for formal methods through simpler, more tailored semantics and syntax. Typical examples of formal specification languages are purely logic-based languages used in theorem provers [45, 92, 26], and guarded command languages used in model checkers [86, 81]. This is the so called *specification gap*.

Similar to the system model, the user also has to formally specify the requirements. Correctly specifying complex properties in temporal logic also requires expertise. Developing easier ways for normal developers with minimal model checking experience to specify these properties is an active area of research, but is considered outside the scope of this thesis. Two attempts worth mentioning is the Timeline Editor of the FeaVer system [74], and the Bandera Specification Language (BSL) [24, 50]. Another issue is the fact that the system is only verified for the stated requirements, and no guarantees can be made about the correctness or completeness of the requirements specification. This is true for any verification technique, though.

2.3.3 The Specification Gap

The first issue raised by the specification gap is the possibility of discrepancies between the behaviour being verified and that of the actual implementation. Since a model of the system is model checked, and not the actual system implementation itself, we are not guaranteed that properties verified to hold for the model also hold for the implementation. Manually constructing a verification model that is true to the actual system, as well as tractable, is a task that for non-trivial systems require skill and expertise not normally associated with software development. Wolper stated that "Manual verification is at least as likely to be wrong as the program itself" [111], and although the actual verification is formal and automated, the possibility of human-error in the translation between model and implementation could still invalidate the verification.

The second issue is that constructing the system model is time consuming, which can be prohibitive when developing non-trivial applications. Time spent on development of course directly translates to cost in the industry. As already mentioned, this process also requires specialised expertise, which again could increase production costs if these skills are not already available in-house. It has, however, been shown that the actual cost of applying model checking is in many cases not that high when compared to other verification methods, and is usually outweighed by the benefits it offers. Unfortunately this remains a perceived issue, which because of the pressure to reduce design and development time ("time to market") is hindering the adoption of model checking in the industry. This is sometimes referred to as the *technology transfer problem* [10, 48]. Perhaps more relevant are the issues of *timing* and *relevance*. In a typical software development environment where teams of developers work on a project, the system design or implementation can change rapidly, even daily. Constructing the model, however, can take a lot longer, up to weeks or even months for a sizable system. This means that either the development is held up waiting for the verification results, or that the results will most likely be irrelevant when they become available since they are based on a previous version of the system.

The purpose of formal methods is to enable the development of correct software applications (i.e., that satisfy their specification), but powerful verification techniques are useless if not adopted by the developers of these applications. Software developers in general prefer to use general program languages such as C, C++ and Java, and are reluctant to adopt development techniques that require use of specialised languages that are also generally less expressive. It is also true that many project managers do not recognise that time spent on model checking is beneficial in the long run, and tend to view it as time not spent on actual system implementation.

This has led to research being directed at making model checking a more attractive option for software development. The idea is that by addressing the *specification gap* and the issues associated with it, model checking can be made easier to use and more efficient at ensuring reliability of the actual system implementation. Current research in this field can be grouped into three directions: In the first group an implementation is generated from a verified specification or model, in the second a verification model is extracted from existent program source code, and in the third the program source is model checked directly. We discuss the techniques, as associated with each group, in more detail in the following Sections, along with some of the more prominent research projects in each group.

2.4 Implementation Generation

Of the three approaches, the first is the closest to the method traditionally prescribed for formal software development. In accordance with traditional software engineering methodology, the system is first specified in terms of its design and requirements, the design verified against the requirements, and the verified design then used to implement the actual system. Although it has been stated many times that this last step of deriving the implementation from the model could potentially be automated, it has largely remained a manual process. Logical refinement techniques during the extraction process can help to preserve the essential correctness properties, but as long as the process is not automated, the specification gap remains a problem. Automating this step addresses both the issues of reliability and cost; one such an attempt is the system by Löffler and Serhrouchni for creating C code implementations from Promela models.

2.4.1 A Promela-to-C Compiler

The approach taken by Löffler and Serhrouchni extends Spin to allow for the generation of an implementation from a Promela specification [82, 84]. As explained in subsection 2.2.4, Spin generates a verifier in ANSI C from the Promela specification. The generated source code for the verifier already contains code for all the transitions and actions; this is isolated and extended with additional code to produce an implementation. This discussion assumes some knowledge of Spin and Promela. For any unfamiliar concepts the reader is referred to the next chapter which contains a detailed discussion of the Promela language.

The additional code implements a pseudo *runtime environment* that replaces the Spin code that drives the state machine. It consists of a scheduler, as well as some code that implements timer and external communication functionality. This additional code is then compiled together with the code implementing the state machine to produce a singe UNIX program. When this program is compiled and executed, the built-in runtime environment schedules the execution of the Promela processes and handles all communication.

Non-determinism is handled by randomly choosing one of the executable branches for the first selection, and thereafter selecting the first executable branch. If a process has no executable next branch, it blocks, and if all the processes are blocked the actual UNIX program is also blocked until an external event occurs or a timer expires. Since version 2.0, Spin allows the system to switch to another process if the current process blocks within an `atomic` sequence (a code sequence that is executed as an indivisible unit). The generated scheduler does, however, not have as much control over the system as Spin, and consequently does not know if a process waiting for an external event will deadlock or not. For this reason the implementation differs from the model behaviour in that a process blocking in an `atomic` sequence will stay blocked until it unblocks itself. This example shows how it is possible to interpret the semantics of Promela differently even when using the same system specification representation.

Both the timer mechanism and external communication are implemented using message channels. For the timer mechanism three channels are provided: `set_timer` is used to set a timer, `timer` is used to listen for a timer expiring, and `del_timer` is used to delete a timer before it expires. Whereas the runtime environment provides this functionality in the implementation, an additional process has to be added to the model for simulation and verification.

External communication channels are defined with the prefix `‘ext_’`. Again, for simulation and verification purposes, additional processes are included into the model to represent the environment. An implementation of this external behaviour is placed in a separate file and included in the main model. An option to generate the Promela specification as separate UNIX processes is available. External communication is based on server-client model. The server handles all the external communication and keeps track of the queue content, handles requests for reads from a channel, and notifies all clients if the content of any external channel changes. The actual communication is implemented using AF_UNIX domain sockets, which can be manually changed to AF_INET sockets for distributed communication.

The usefulness of this extension to Spin is limited, and the authors themselves describe its application field as the creation of test scenarios and the rapid prototyping of validated protocol implementations. One example is the generation of an implementation for the “Steam

Boiler Control Specification Problem" [83].

2.4.2 Domain-Specific Languages

The dilemma facing any attempt to automate the generation of implementations is that, since system models should be kept as simple as possible for model checking, the specification generally does not contain enough information to allow the automated derivation of a detailed and efficient implementation. *Domain-specific languages* address this issue by optimising the process for a specific application area. Many of these languages are supported by compilers that can generate an application from the specification.

One such language is Esterel [8], which was designed for specifying synchronous reactive systems. The Esterel compiler generates a unique automaton from the specification. In the specific application area of protocol development, the HIPPCO optimising compiler [16] was developed to generate efficient protocol code from a protocol specification written in Esterel. Just like Promela, Esterel can be considered to be insufficiently expressive, which led to the development of Promela++ [6, 7, 5].

Promela++

Developed as part of a language-based framework for protocol construction, Promela++ is based on Promela and has been designed with a rich set of domain-specific constructs to facilitate protocol construction. This enables the Promela++ compiler to perform domain specific optimisations. The compiler can automatically convert protocol specifications in Promela++ to protocol models in Promela for verification, or to efficient C code for execution.

In a Promela++ program the control structure is separated from the low-level network access and data manipulation routines. This is done by specifying the control structure in Promela++, which can be converted directly to Promela, and implementing the low-level routines as blocks of C code embedded in the program. The C code that is allowed is restricted, specifically in the use of pointers, but still provides adequate functionality, especially in the form of access to system resources. When a Promela model is generated from a Promela++ specification the C code is not included and the model represents a finite-state automaton that encapsulates the control flow.

Promela++ has been used to implement protocols such as Horus and TCP, and generate code with comparable performance to hand-optimised monolithic implementations. Unfortunately, Promela++ is restricted to the implementation of layered communication protocols.

2.4.3 A Transition System Approach

Bull and De Villiers have proposed a more general approach for the generation of reliable reactive systems, namely separating the flow of control from data manipulation [15, 14]. Using the same structure for the system model and implementation can help to ensure that the benefits obtained from the verification of the model is carried over to the implementation code. The authors suggested the use of *transition systems* as a general structure for reactive systems.

A transition system is identical to a finite-state automaton, but without any notion of acceptance [1]. A transition system can be represented as a graph, with the graph vertices representing the unique system states and the edges representing the transitions between the states. Each transition has a finite set of events associated with it, and the occurrence of any of these events enables the corresponding transition. Only enabled transitions are executable. A unique action is executed in response to each event, where actions are small independent code fragments that implement operations that may modify the system state.

Because reactive systems are usually event-driven, they naturally lend themselves to be structured as transition systems. Furthermore, transition systems are also suited to automata-theoretic model checking (as described in Section 2.2).

While the flow of control is model checked, it is still necessary to verify the details of data manipulation, but this is significantly simplified by the structure of the transition system. Since data manipulation is restricted to the separate, independent code segments that execute independently of unpredictable external events, their correctness can be ensured through the use of sequential program verification techniques such as weakest preconditions described in Section 2.1. These data manipulation code segments can also be optimised for the implementation without affecting the verification of the control structure.

Bull and De Villiers used this technique to implement a version of the alternating bit protocol with assembly level coding, and a minimal implementation of the TCP protocol in C as an example of a larger application. The TCP implementation turned out to be less efficient, but the authors claim that the problem is not inherent in the technique. More importantly, however, is the fact that the implementations were done by hand, which means that timing and relevance still remain issues, as well as the (albeit reduced) problem of reliability caused by manual derivation. Again, automating the process of deriving the implementation from the verified model would address these issues, and the benefits of this approach would be preserved.

2.5 Model Extraction

Where in the first approach model checking is applied during the analysis and design phase, the approach described here is applied further down the software development process. A model is constructed from an actual implementation, the model verified, and the verification results applied to the implementation. The use of abstraction techniques is usually necessary to construct a tractable model from an implementation. This approach originates from investigations into the application of model checking to verify existing applications. In the first case studies the models were manually derived from the implementation code, but it quickly became apparent that for this approach to be viable the extraction process would have to be automated. This led to program model checking based on automated model-extraction, where the program is translated into the input notation of an existing model checker.

2.5.1 Manual Approach

The use of SPIN to support the automated verification of time partitioning in the Honeywell DEOS real-time scheduling kernel [96, 97] is a good example of the manual application of this approach. The goal of the experiment was to investigate whether model checking with minimal abstraction could be used to find a subtle implementation error that was originally discovered and fixed during the standard formal review process.

To apply model checking, a core slice of the DEOS (Dynamic Enforcement Operating System) scheduling kernel was manually translated from C++ to PROMELA. To close the system for verification an abstraction of the system environment also had to be constructed. This was found to be the most difficult task in the original experiment. It was also found that program abstraction techniques are critical in making verification tractable.

The experiment was successful in rediscovering the known error in the implementation, showing that model checking could find errors in real programs that testing techniques did not. After the original experiment the model was expanded to bring it up-to-date with the kernel implementation. This update was done in one day by a Honeywell developer with no prior knowledge of PROMELA or SPIN, and on the first model checking run after the update a new error was discovered by Spin. The update of the model was based on a slightly outdated version of the DEOS code, and the error had already been discovered by Honeywell the previous week. However, where it originally took three days to discover the cause of the error, with Spin it was possible to easily replay the error trace and identify the error.

A similar case-study was the application of Spin to analyse part of the Remote Agent space craft controller [52]. This effort revealed four previously unknown classic concurrency errors. A very interesting development was that when the Remote Agent was finally activated in space it suffered a critical malfunction, which was later identified to have been caused by a coding error similar to one of those found using Spin. This was in a part of the system not analysed in the case study, and it demonstrated to NASA that model checking can successfully be used to find mission critical errors.

These experiments clearly showed that model checking could be applied to implementation level code, and was successful in identifying critical errors that easily slip by traditional testing techniques.

2.5.2 Java PathFinder I

Java PathFinder was developed as a first attempt by the Automated Software Engineering group at NASA Ames to automate the process of using model checking to analyse programs written in traditional programming languages [51, 53]. They chose to develop the system for Java, since they wanted to base it on a programming language that was object oriented, as well as popular. The tool was renamed to Java PathFinder 1 (JPF1) after the Mars PathFinder rover that explored Mars in 1997 [54].

JPF1 translates a given Java program into a Promela model, which can then be model checked using Spin. The model is generated with the same state space characteristics as the Java program (no analysis is applied to reduce the state space), which restricts the application to Java programs with finite and tractable state spaces. The Java programs may contain assertions that are translated to Promela assertions. Spin can then be used to check the model for deadlock and assertion violations. It is of course also possible to use Spin to check general LTL properties.

JPF1 only accepts a subset of Java 1.0, which restricts the possible program input. The authors do note, however, that at the time it translated more of Java than any other similar tool. A bigger issue is that certain features such as recursion and, more importantly, predefined class libraries are not supported. This severely restricts the applicability of the tool.

After this first implementation, it was realised that these restrictions could be avoided by instead applying model checking on the byte-code form of a Java program, without incurring a big loss in efficiency. This led to the development of Java PathFinder 2 (described in Subsection 2.6.4).

JPF1 was, however, considered a successful first experiment. It was used to detect a deadlock situation in a game server with 1400 lines of code, as well as to successfully locate the error that caused the malfunction of the Remote Agent (described above). A similar project is the JCAT tool that also generates Promela models from Java source code, for verification with Spin and dSpin [30].

2.5.3 Bandera

Bandera is a tool for model checking Java source code, and implements a component-based architecture for model extraction designed to maximise scalability, flexibility and extensibility [25, 33]. While it is similar to the JPF and JCAT tools in the sense that it also translates Java source code to the input language of one of several existing model-checking tools (which includes Promela), it is much more than just a translator, and specifically differ from these tools in that it implements techniques that allow the generation of verification models tailored to the property to be checked. The Bandera toolset is an integrated collection of program analysis, transformation and visualisation components. It takes as input Java source code and a software requirement formalised in Bandera's temporal specification language, and generates a program model specification in the input language of one of several existing model-checking tools, which includes Spin [61], dSpin [29], SMV [85], and Java PathFinder [12].

Bandera generates a custom program based on the specific property to be checked through the use of program slicing and abstraction. A library of abstractions are available for the user to choose from, or optionally add new abstraction definitions to. The reduced representation of the program is fed to the Bandera backend, which generates a finite-state model in the input language of a verification tool as chosen by the user. The output generated by the verifier is then interpreted by Bandera, and any counterexample is mapped back to the original source code. Bandera also allows the user to inspect the error trail interactively.

Although Bandera can handle realistic classes of Java programs, it accepts a restricted set of Java. Several other limitations are described in [50], but it should be noted that Bandera is not so much an industrial-strength tool, as a large collection of diverse tools to facilitate further experimentation. A similar tool was developed by the Hardware Verification Group at

Stanford University [93]; it translates Java to SAL, an intermediate language for interfacing model checking and theorem proving tools.

2.5.4 FeaVer

FeaVer is a software verification system for distributed software systems implemented in ANSI C [63, 64, 74]. At the heart of the system is a method to mechanically extract verification models from source code using a system of abstractions. The extractor generates a PROMELA model that faithfully reproduces the control structure of the C code implementation. If no additional user input is provided all basic statements and expressions are encapsulated in `c_code` and `c_expr` PROMELA primitives. (Subsection 3.1.12 explains these primitives in more detail.) The user can refine the extraction by providing a lookup-table that describes conversion rules, matching C source code fragments with corresponding model representations. This makes it possible to define syntactic conversions, as well as a broad range of general abstraction rules. To close the system for verification, the user additionally has to specify the execution environment the system interacts with (similar to test-drivers), as well as specify the formal requirements. Once this is done, SPIN is used to perform the actual verification.

This method effectively replaces the cost of manually constructing a verification model for each significant version of the source code, with that of defining an initial lookup table and keeping it up-to-date to track changes in the source code. The model extraction tool notifies the user if it detects any omissions, mismatches or redundancies between entries in the table and statements in the source code. This allows the verification effort to follow the product through every phase of the design cycle, verifying virtually every version of the source code, without disrupting the normal development process. The entries in the lookup-table also explicitly document the conversion rules.

FeaVer was originally developed at Bell Labs as a feature verification system (hence the name) for the call processing code of the the PathStar® Access Server of Lucent Technologies [72, 73]. It was used between 1998 and 2000 to exhaustively verify the call processing software, and this effort is generally regarded as the first application of software model checking on a large scale in a commercial setting. Significantly, the technique revealed ten times as many serious software defects in the target code than conventional testing did. This included the early detection of undesired feature interaction, which prompted a rethinking of the code structure and avoided an expensive redesign later on. The system also proved useful as a diagnostics tool. It was able to identify and reproduce system behaviour corresponding to errors encountered, but not reproducible, during testing. Additionally, the system was also used to confirm suspected behaviour, and identify unreachable code.

The original project gave rise to a standalone software verification tool, which accepts unrestricted ANSI C, with the reliance on a special source code notation removed. The name FeaVer that was used to describe the feature verification system now describes the user interface. These include the model extractor Modex, the logic verifier Spin, a C preprocessor and compiler, and a couple of Unix-style postprocessing commands. This tool has been applied to a range of commercial products, ranging in size from a few hundred to 160,000 lines of C source text, and in all cases concurrency related errors were found in the source code that had escaped routine conventional system testing [64].

2.6 Program Model Checking

In the previous section we described an approach where verification models are mechanically extracted from an application's source code to apply model checking at the implementation level. The approach described in this section takes this one step further, by implementing dedicated model checking tools that interpret the programming language directly. This has the benefit that the source code does not have to be translated to a model checking language with different syntax and semantics. This approach also has to deal with the added complexity that follows from using a general programming language as the input language for the checking tool. Currently there are several projects that focus on model checking programs written in C, made possible through the use of heavy abstraction and limiting the information stored during exploration, as well as a project with the novel approach of model checking Java programs at the byte-code level.

2.6.1 Specialised Implementation Languages

One way to make the process of model checking programs at the implementation level easier, is to use programming languages that lend themselves more readily to being model checked (i.e., that are closer to the input languages accepted by model checking tools). An example of this approach is the dedicated model checker implemented by Huch [75, 76] for a subset of Erlang, an untyped higher-order soft realtime, declarative, functional language for concurrent, distributed systems (designed at the Ericsson Computer Science Laboratory and recently open-sourced: `http://www.erlang.org/`). This kind of approach is unfortunately limited by the fact that these more specialised languages are not as popular with developers in general, which is why we focus on more general programming languages such as Java and C.

2.6.2 VeriSoft

The VeriSoft tool was developed by Godefroid at Lucent Technologies, for systematically exploring the state-space of a concurrent system composed of processes described by programs written in C or C++ [40, 43, 44]. The tool addresses the complexity of model checking actual programs by doing a state-less search, and applies *partial order methods*, in the form of sleep sets and persistent sets, to reduce the number of revisited states.

Every process of the concurrent system is mapped to a UNIX process, and the execution of this new system of processes is controlled by an external process called the `scheduler`. This control includes being able to suspend the execution of a process. The code of the processes is instrumented to make the control points visible to the `scheduler`, which uses these visible operations to drive the system through the states and transitions of its state space.

A *transition* is defined as the execution of a process from one visible operation to just before the next visible operation, possibly including a finite sequence of invisible operations, and the system is said to be in a *global* state when for every process in the system the next operation to be executed is a visible operation. It is also assumed that only execution of visible operations may be blocking. By resuming the execution of one of the system processes in a global state, the `scheduler` can explore one transition at a time.

As with most other model checking tools, the system needs to be closed for verification. This is done by specifying additional programs or test-drivers. VeriSoft provides special mechanisms with which non-determinism and requirement specifications (such as assertions) can be added to the test drivers.

VeriSoft drives the closed system through all possible execution paths in search of deadlock and assertion violations. To prevent the search from becoming trapped in a cycle, the search depth of the search is limited. When the maximum depth is reached, the scheduler re-initialises the system and explores alternative paths in the state space. The search is stopped as soon as an error is detected, and the execution trace formed by all the transitions stored during the current exploration is output. VeriSoft provides an interactive graphical simulator that enables the user to view the execution trace at the instruction or procedure level.

Some of the benefits of the VeriSoft approach are that it is versatile, in that it does not rely on any specific assumption about the code representing the process behaviour, and that it is scalable, since its applicability depends on the amount of non-determinism of the system being analysed, and not the size of the code. One limitation is the fact that a state-less search cannot detect cycles, and is thus restricted to the verification of safety properties.

VeriSoft has been applied successfully for analysing several software products at Lucent Technologies, including the analysis of the "Heart-Beat Monitor" of a 4ESS switch, a critical component of a telephone switch, and several releases of call-processing software running on Lucent's CDMA base-stations, a multi-billion dollar product line. In one of its first applications it successfully discovered an error in a 2500-line C program controlling robots in an unpredictable environment.

Similar projects include the state-less model checker for Java programs developed by Stoller [104], and the CMC explicit state model checker for ANSI C programs by Musuvathi [91]. Stoller generalises the work done by Godefroid on VeriSoft in order to handle multi-threaded programs, as well as the work done by Bruening on ExitBlockRW [13] to handle multi-process systems. CMC also generates the state space by directly executing the implementation, but contrary to VeriSoft does store the system states. Instead, the tool applies aggressive compression techniques to capture as much state information as possible.

2.6.3 Slam

SLAM is a research project at Microsoft, that checks whether Windows XP device drivers obey the operating system's API usage rules [3, 105]. It is applied to sequential C programs, which it statically checks against safety properties, using techniques from program analysis, model checking, and automated deduction. The properties are encoded in SLIC, or Specification Language for Interface Checking.

The basic operation of the system consists of iterating the creation, analysis and refinement of program abstraction, until either a feasible execution path in the program that leads to a property violation is found, the program is validated (no path exists that reaches a property violation), or the process runs out of resources. Showing that a path is feasible or not may turn out to be undecidable, in which case the checker returns a "don't know" result.

SLAM uses *predicate abstraction* to create a sound Boolean program abstraction of the C program, where a Boolean program contains the control structure of the original program, but with all other information abstracted as Boolean predicates. If reachability analysis determines that the special label ERROR (property violation) is not reachable in the Boolean program, then it is also not reachable in the original program. The process, however, starts by abstracting the program to its most over-approximated representation, so it is possible that violation paths are detected in the abstraction that do not exist in the original. In this case *counterexample-driven refinement* is applied to create a more precise Boolean program by removing the spurious path from the abstraction through the introduction of new predicates. This will also remove any other spurious execution paths that include branch options removed by the restriction. The process is then repeated with the refined Boolean program.

The SLAM toolkit consists of the SLIC preprocessor, the C2BP tool that performs the predicate abstraction, the BEBOP model checker [2], and NEWTON, a tool that discovers additional predicates to refine the Boolean program. In [105] it is stated that the toolkit has been used to successfully analyse programs on the order of 10,000 lines, and abstractions with several hundred Boolean variables in the range of minutes to a half hour.

A similar project is BLAST, or Berkely Lazy Abstraction Software Verification Tool [56]. BLAST is a software model checker for C programs, and has the goal of checking that a software application satisfies behavioural properties of the interfaces it uses. It uses counter-example-driven automatic abstraction refinement to construct the abstract model according to the requirements. It differs from SLAM in its use of *lazy abstractions* [55] to short-circuit the abstraction–verification–refinement cycle, tightly integrating the three steps.

Another project is CBMC [18, 17], a tool for the formal verification of ANSI C programs using *bounded model checking*. In short, this approach to model checking converts a partial state space to a Boolean formula that is satisfiable if and only if the system contains a violation. The partial state space is gradually refined, until a point is reached at which the tool can guarantee that no further refinement is needed. For a more detailed description, interested readers are referred to [9].

2.6.4 Java PathFinder II

After JPF1 (discussed in Subsection 2.5.2), the ASE group at NASA Ames developed Java PathFinder 2, or JPF2 [11, 12, 47, 54, 110]. Although JPF1 was seen as successful it suffered from a number of drawbacks, which resulted mainly from the fact that the approach was based on translating Java to PROMELA. The mapping between Java and PROMELA is not straightforward, and meant that JPF1 could not (1) handle all the language features of Java, (2) handle Java libraries, and (3) allow more flexible approaches to model check Java programs. JPF2 was developed specifically to address these shortcomings, and does so by model checking the byte-code representation of the application.

JPF2 is based on a Java Virtual Machine, or JVM, implemented specifically for model checking, which means that it was designed for efficient memory management rather than execution speed. This JVM can execute all Java byte-codes, which in effect means that JPF2 can handle all Java language features, as well as all Java libraries, since everything is compiled to byte-code. This addresses the first and second issues. The third issue was addressed by designing JPF2 to be modular and thus allow different search strategies to be easily integrated into the model checker.

Whereas some other approaches, such as VeriSoft, use state-less model checking to handle the additional state-space requirements caused by the complexity of programs written in general programming languages, JPF2 is an explicit-state model checker. The system states generated for a Java program can be very large, severely restricting the size of systems that can be model checked. JPF2 therefore applies aggressive state compression techniques that reduce the memory requirements of the model checker by an order of magnitude. The tool also includes symmetry reduction to take advantage of the fact that object oriented programs typically make use of many objects, many of which can be considered equal during verification. Furthermore, JPF2 supports distributed memory model checking. This allows the memory requirements of the model checker to be shared over a number of workstations, and although it introduces additional runtime overhead, it allows the verification of programs that exceeds the physical memory of a single workstation.

Instead of reimplementing a complete user interface from scratch, JPF2 makes use of the Bandera toolset for specifying the properties to be analysed (in the Bandera Specification Language, BSL [24]), and to display counterexamples as an error trail in the source. JPF2 also makes use of Bandera's functionality to apply certain forms of abstraction and slicing. BSL is fully integrated into JPF2. In order to also accept requirements properties specified in LTL, JPF2 has a front-end translator from LTL to Büchi automata.

JPF2 has been successfully used in a number of projects. This includes a reanalysis of the DEOS kernel (described in Subsection 2.5.1), and the successful analysis of 7,000 lines of code from a Mars rover. JPF2 was open-sourced in February 2001 (`http://javapathfinder.sourceforge.net/`). The tool has also realised one of its main goals of becoming a vehicle for model checking research; the ASE group works in close collaboration with the Bandera group at Kansas State University, as well as other groups at CMU, Stony Brook, Minnesota, Freiburg, and Liverpool Universities.

Chapter 3

Implementation Generation

The ideal design method advocated by many in the model checking community consists of formally specifying the design of an implementation as an abstract model, proving it correct through verification, and then extracting an executable software implementation from the verified model [72]. Since a model, per definition, contains less detail than the implementation, it is probably more accurate to say that an implementation is generated, rather than extracted, from the design.

As we saw in the previous chapter, one drawback of this approach is the final step of manually extracting a software implementation from the verified system model. This thesis presents a technique that addresses the issues associated with this step by formalising and automating the generation of an executable software model implementation. This chapter describes the approach, including the design and implementation of the program that automates the generation process.

Formalising the implementation generation ensures that the implementation conforms to the model specification, so that the properties verified for the model also hold for the implementation. By "formalising" we mean that the steps taken to transform the model into an implementation is formally defined and verified to be logically sound. Automating the process minimises the human factor, which greatly reduces the chance of errors in applying the translation steps and speeds up the process. This technique reduces the cost and increases reliability.

Using this suggested design method implies that the system design is specified in enough detail so that a complete software implementation can be generated automatically. Since Spin [61] has already been proven effective at verifying production class software, and its input language Promela is closer to a programming language than other specialised specification languages, it was decided to use Promela as the specification language. Even though Promela resembles C, it was designed for the specification of abstract models of communication protocols, and not as an implementation language. On the contrary, it was designed to suppress implementation and programming detail. The dilemma here is that we want to generate an efficient implementation from a Promela specification, while still preserving the attributes of the language that allow the use of the Spin system for simulation and verification. As a compromise, it was decided to extend Promela to allow the specification of a system with enough information to support the fully automated generation of an implementation program source, but then also generate a version of the specification that consists of only standard Promela.

For the implementation it was decided to generate the program source in ANSI C. Promela already has a strong connection to ANSI C because of its C-like syntax, and from version 2.0 onward the Spin source code, as well as all programs generated by Spin, conforms fully to the ANSI C standard. Since these sources are freely and readily available they offer an insight into the translation of Promela to C. Also, Promela supports the inclusion of embedded ANSI C code fragments as part of the FeaVer model extraction project [73]. The second reason is that ANSI C is a very popular and highly portable programming language. This way we ensure a wide application area and hopefully also user base by supporting a programming language favoured by many as the implementation language of choice for the very systems that are ideal candidates for this verification technique.

As Section 3.2 describes in detail, Promela was extended to allow the user to specify **model specific** and **implementation specific** code fragments. This amounts to defining abstract–concrete mappings that are used to generate either the abstract system model or the concrete implementation. Apart from the mappings, Promela control flow structures are used to describe the high-level control flow of the system. This means that the Generator must be able to map Promela code to C code — a process described in detail in Section 3.1.

As a result, data manipulation is confined to small, independent code segments that specify system events and actions. This has the added benefit that data manipulation, which does not naturally lend itself to verification through model checking, is defined in self-contained segments that can be verified by specifying pre- and postconditions and applying traditional testing techniques.

3.1 Translating Promela to C

This section describes the main features of the PROMELA language, and, importantly, shows how they are translated to ANSI C. The definition of the PROMELA language elements given here is based on the SPIN online references, available at `http://spinroot.com/spin/Man`. The following EBNF-like notation is used to describe the PROMELA grammar:

- Choices are separated by vertical bars: `|`
- Optional elements are included in square brackets: `[...]`
- A Kleene star, `*`, indicates zero or more repetitions of the preceding fragment.
- Literals are enclosed in single quotes: `'...'`.
- Uppercase names refer to tokens, or terminals, representing keywords, while lowercase names refer to grammar rules.

The main components of a PROMELA specification are processes, message channels and variables. Within a PROMELA specification the processes are global objects, while message channels and variables can be declared either globally or locally to a process declaration.

The most obvious way to model a system in PROMELA is to represent each of the processes in the system being modelled, including the environment, with a PROMELA `proctype`, use the message channels to represent inter-process communication, and the local and global variables for local process and global, shared system memory respectively. However, PROMELA can and has been used to model system behaviour in a lot of different ways. For example, a PROMELA `proctype` can be used to model a function call, or two or more proctypes with global variables can represent threads executing within a process. Since it would be impossible to automatically generate an implementation from such a very abstract model without additional information or guidance, the default behaviour of the GENERATOR is to translate the PROMELA language elements to the closest real-world representation. This will be defined more clearly for each element.

3.1.1 Metaterms

Metaterms do not add functionality to the language, but rather provide a more convenient mechanism for specifying a model. Comments, for example, allow a user to specify additional information that could make the PROMELA specification easier to read, without changing its meaning.

To support metaterms the Generator, similar to Spin, first preprocesses the system specification. The standard C preprocessor **cpp** is used to handle all ANSI C metaterms, which include comments, macros and include files. This step is invisible to the user. Named sequences, or inlines, are specific to Promela and are handled internally by the Generator. Because the source file is first preprocessed using **cpp**, the Promela only file that is generated will already have all C preprocessing done on it.

3.1.2 Inline Functions

```
INLINE name '(' [ arg_lst ] ')' '{' sequence '}'
```

An `inline` is a stylised version of a macro, and defines a replacement text for a symbolic name, possibly with parameters. The benefit of using an `inline` function is that during simulation the line number references are preserved by Spin and indicates the source line inside the `inline` definition, which is not the case when using a C macro. An `inline` has to be defined globally, and before its first use in the specification. The syntax of an `inline` call, or invocation, is similar to that of a procedure call in C, but the actual semantics are defined by the body of the `inline`. An `inline` definition can contain other `inline` calls, but may not call itself recursively.

An `inline` call can appear anywhere in the model where a stand-alone statement is allowed, which means that unlike a C macro it cannot appear in the parameter list of a `run` statement or used as an operand in an expression. It is also not allowed on the left or right of an assignment statement.

It is also important to note that `inline` definitions do not affect Promela scope rules. The scope of a variable declared within an `inline` is defined by where the `inline` is called; if invoked within a process declaration, its scope is the body of the process.

3.1.3 Symbolic constants

```
MTYPE [ '=' ] '{' name [ ',' name ] * '}'
```

In addition to macros, Promela also provides the `mtype` declaration for the introduction of symbolic names for constant values. There can be multiple `mtype` declarations in a verification

```
mtype = { const1, const2, const3 }

mtype = { const4, const5 }

mtype = { const6, const7, const8 }
```

```
enum mtype {
    const3 = 1,
    const2,
    const1,
    const5,
    const4,
    const8,
    const7,
    const6
};
```

Figure 4: Translation of an `mtype` definition.

model. However, unlike macros that are replaced with their actual values by the preprocessor, the `mtype` constants remain in symbolic form.

Implementation

If a model contains any `mtype` declarations, the GENERATOR declares a C `enum` variable named `mtype` in the implementation and adds equivalent symbolic constants to it to match that of the models' `mtype`. The enum declaration is placed in the `init.h` header file so that all the generated programs have access to the constant definitions, just as in the model.

For consistency the symbolic constants are added to the enum in such a way that the constant value assigned to each symbol is the same as in PROMELA. Because of the somewhat odd way that this is done in PROMELA, the arrangement of the symbolic constants is not what one would expect. In PROMELA, for every mtype declaration the symbols are numbered in the reverse order of their declaration, and the lowest number assigned is one, not zero. If multiple `mtype` declarations appear in the model, each new set of symbols is prepended to the previously defined set. The example in Figure 4 illustrates the translation.

3.1.4 Processes

The behaviour of a process is defined using a `proctype` declaration, which has the syntax:

```
[ active ] PROCTYPE name ‘(’ [ decl_lst ] ‘)’
[ priority ] [ enabler ] ‘{’ sequence ‘}’
```

where the non-optional terms in the declaration are:

- `PROCTYPE`, which is usually the keyword `proctype`, or optionally the keyword `D_proctype`. The `D_proctype` keyword is used to declare a process with deterministic behaviour, where the default behaviour for a PROMELA process is non-deterministic.
- `name = letter [ letter | number ] *`, a unique process name.
- `'(' ')'`, enclosing brackets for the optional formal parameters.
- `'{' sequence '}'`, the body of the process declared as a sequence of declarations and statements enclosed in curly braces.

The optional terms in the declaration are:

- `decl_lst = one_decl [ ';' one_decl ]`, the formal parameters of the proctype. Formal parameters provide a way to pass arguments to the process at instantiation. They are only allowed to be of a basic, or message channel type. Within the `proctype` body formal parameters are seen as part of the local variables, the only difference being that their initial values can be set at instantiation. The actual parameters are pass-by-value.
- `active = ACTIVE [ '[' const ']' ]`, a prefix which is used for process instantiation, and will be described in more detail along with process instantiation.
- `priority = PRIORITY const`, a setting that can be used in random simulation to change the probability that the process is scheduled for execution. Since it is a simulation option, it doesn't translate to the implementation.
- `enabler = PROVIDED '(' expr ')'`, an option for setting a global constraint on process execution. It is also primarily used for simulation, and for this reason it is not currently supported by the GENERATOR.

A minimal process declaration can be specified as follows:

```
proctype A() { skip }
```

where `A` is the name of the `proctype` and `skip` is a dummy, null statement (since the body is not allowed to be empty). Having described the basic syntax of a PROMELA process, we can now define how it will be represented in the generated implementation.

Process Instantiation

A `proctype` definition only declares process behaviour. To execute this behaviour an instance of the process has to be spawned. It is thus possible to execute multiple instances of a certain process. Processes in PROMELA are either instantiated automatically at system startup, or dynamically by an executing process. Once instantiated, a process executes concurrently within the operational model of the language.

PROMELA provides the `run` unary operator for dynamically creating processes.

```
RUN name ‘(’ [ arg_lst ] ‘)’ [ priority ]
```

The `run` operator takes as argument the name of a previously declared proctype, as well as actual parameters which match the formal parameters of the particular `proctype` declaration. A `run` statement is only executable if a process of the type specified can be instantiated. This may not be possible, for example, if there are already too many running processes. SPIN (arbitrarily) limits the number of running processes to 255. If the operation is successful it returns the process instantiation number, or `pid`, of the newly created process. If it fails zero is returned.

The Initial Process

The execution of a model must start *somewhere*, and so a PROMELA model must have at least one active process in its initial system state.

The `init` keyword — which is used in the place of the `proctype` keyword— declares such a process. This process takes no parameters and there can only be one instance of it active in the model, the initial instance. This means that it cannot be instantiated using the `run` operator. The `init` process has traditionally been used to initialise the modelled system by initialising global variables and instantiating other processes. The most basic functional PROMELA model one could specify consists of only the `init` process, with only the simplest system possible behaviour:

```
init { skip }
```

```
proctype A() {
    skip;
}
```

```
main() {
    1;
}
```

Figure 5: Translation of a minimalist `proctype` declaration.

Active Processes

It is also possible to instantiate an initial set of processes using the `active` prefix for `proctype` declarations. This specifies that an instance of the `proctype` should be active in the initial system state. The `active` prefix also has an optional integer suffix to specify more than one instantiation. When a process is instantiated using the `active` prefix, it is not possible to pass its arguments, so in this case the formal parameters are all initialised to zero.

Implementation

For each `proctype` declaration a C program source is generated in a file with the name of the `proctype`. The body of the process is simply generated within a **main()** C function. As an example, the implementation source for `proctype A` in the example

```
proctype A() { skip }
```

would be generated in a file named `A.c`, with the translation shown in Figure 5.

The `skip` statement is simply replaced with 1, or true (which is a redundant, but direct translation). This source file is compiled using an ANSI C compiler to create a program with the same name. For this example a program with the name `A` would be generated. Executing this program is then equivalent to instantiating a version of the process in the model. Since each `proctype` in a model has to be uniquely named, we can be sure that the implementation source will have unique file names (per application of course).

In PROMELA the formal parameters of a `proctype` are seen as local variables inside the body of the process. In the implementation, the parameters are declared as local variables in the `main` function, and their values are derived from the program's command-linearguments. This accurately mimics the actual, pass-by-value parameters. Figure 6 provides a simple example that illustrates this process.

```
proctype add(int x; int y)
{
    int z;

    z = x + y;
}
```

```
int main( int argc, char *argv[] )
{
    /* Parameters: */
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    /* Declarations: */
    int z = 0;
    /* Statements: */
    z = x + y;
}
```

Figure 6: Translation of `proctype` formal parameters.

To make the implementation code easier to read the GENERATOR inserts comments that clearly indicate the parameters, declarations and statements. As can be seen in the example, the standard UNIX **atoi()** function is used to convert the arguments from strings to the appropriate types. Since process instantiation is a relatively rare event, the performance penalty incurred by this step is not serious.

A library function was implemented to provide the same functionality as the PROMELA `run` operator. For consistency, the function was also named **run**, and takes as arguments the name of the process to instantiate as well as the actual parameters to pass to this process.

```
int run( char* process, char* fmt, ... )
```

If successful the function returns the process ID of the newly created process and zero otherwise, the same as the PROMELA version. This means it can also be used as a Boolean condition in the same way as the PROMELA `run` operator.

The function was implemented to work on a Linux system. It uses the POSIX compliant **fork()** function to spawn a new child process, and then uses the Linux **execvp()** function to execute the program generated for the particular `proctype` and replace the process image of the child process with this new process image. The **execvp()** function accepts arguments for the new program in the form of an array of pointers to null-terminated strings, so the actual arguments have to be converted to string form. Since the value of some arguments are only known during execution, this conversion has to be done on-the-fly. To handle this cleanly, the **run()** function accepts a variable argument list. This was implemented using the standard `stdarg.h` library, which conforms to ANSI C. The `fmt` parameter is simply a string of letters that specifies the number and types of optional arguments, and is followed by a comma separated list of the actual parameters for the program. This is similar to how the C **printf()** function handles optional arguments. A sample Promela `run` statement

```
run A(1, x);
```

would then be translated as

```
run(A, "d, d", 1, x);
```

where `A` is the name of the process, `‘‘d, d’’` is the fmt argument that specifies the number and type of the arguments to follow, and `1` and `x` are the actual arguments. The `fmt` string in this example specifies two arguments, both of type ‘d’. The characters ‘d’ and ‘i’ can be used interchangeably to specify the integer type. Currently all `proctype` formal parameters can be represented in integer form, so this is currently the only option implemented.

The implementation for the `init` process is generated in much the same way as for all other processes. The body of the proctype declaration is generated in a C source file named `init.c`. Where the implementation differs from that of other processes is that a header file, `init.h`, is also generated for the `init` process. It was decided to use this header file as a way to include all external libraries needed for the implementation, as well as a way to share global declarations between the implemented system processes. Therefore this header file is included into every process's program source file.

Active processes are instantiated in the order in which they are declared in the model, but are all active at the initial system state. This is translated into the implementation by prepending a `run` instantiation statement for each of the active processes to the front of the statement sequence of the `init` process. These statements are in the same order in which the `active` processes are declared, so they will be instantiated in the same order. Consequently, an implementation will be generated for the `init` process, even if no `init` proctype was declared in the model. The **init** program then serves much the same function for the implementation as what the `init` process originally did for Promela models, initialising the system. Figure 7 shows an example of the `init` process implementation that is generated for a model containing only `active` process declarations and no `init` process.

The first `proctype` declaration defines a process ‘`A`’ of which one instance should be active in the initial system state. This is translated as the first **run** statement in the source code generated for the init process, on line four. This process has two formal parameters, integers `x` and `y`, and we can see how, according to the default specified behaviour, they are initialised to zero by passing default (zero) arguments in the **run** statement. The second

```
1   active proctype A(int x; int y)
2   {
3       printf("Active A!\n")
4   }
5
6   active [4] proctype B()
7   {
8       printf("Active B!\n")
9   }
```

```
1   int main( int argc, char *argv[] )
2   {
3       int _NUM_INST_ = 0;
4       run( "./A", "d, d", 0, 0 );
5       for ( _NUM_INST_ = 0; _NUM_INST_<4;
6                       ++_NUM_INST_ ) {
7           run( "./B", "" );
8       };
9   }
```

Figure 7: Instantiation of `active` processes.

`proctype` declaration uses the `active` prefix with the optional integer suffix, indicating that four instances of process '`B`' should be active in the initial system state. If the Generator detects that there is at least one active process declaration for which multiple instantiations has to be done, the `_NUM_INST_` integer variable is added to the implementation of the `init` process. As can be seen in the example (lines 5–8), this variable is used in a **for** statement to instantiate the active processes.

A key advantage of this approach to process declaration and instantiation is that either (1) the **init** program can be invoked directly, or (2) some or all of the process programs can be invoked separately. This makes no difference to the system model: Spin verifies all execution interleavings, which also includes the actual execution of the implemented system. This is exactly the reason why model checking can detect timing issues that result from the use of concurrency. It is important to remember that this difference between model and implementation will be true for any other method of implementation; the scheduling of processes depend on the underlying operating system.

3.1.5 Variables, Message Channels and Scope

Data objects in Promela consist of variables and message channels, and can only be referenced or manipulated by processes executing within the model. The scope of a data object is defined by its declaration relative to to the process declarations. If declared within the body of a `proctype`, it is local to that process. If declared outside of all `proctype` declarations, it is global.

A global variable or message channel declaration defines one instance of the declared object, which is accessible by all processes in the system. Each instantiation of a `proctype`, however, creates new instances of its local data objects, which are private to that particular instantiation. Each declaration can also optionally include an explicit initialisation field, without which the data object is initialised to zero. Message channels, however, are more like pointers, and have to be initialised explicitly before they can be used. It is possible for example to assign an initialised channel to an uninitialised channel variable.

As we have already seen in Subsection 3.1.4, the formal parameters of a `proctype` are also seen as local variables inside the body of the `proctype`. If the process is instantiated using the `run` operator, the formal parameters are initialised to the actual arguments specified, and if instantiated with the `active` prefix, they are initialised to zero.

Although variable declarations can physically appear anywhere in the body of the `proctype` declaration, they are interpreted by Spin as if declared at the start of the body, before any of the other statements. This can cause unexpected results when the initialisation of a variable uses an expression that contains a reference to another local variable.

Implementation

Spin's treatment of declarations (moving them to the start of the process declaration) corresponds to that of ANSI C. To accommodate this, the Generator maintains a separate structure for declarations and statements during parsing.

Promela message channels are used to model a wide range of mechanisms. For this reason it is meaningless to define a default translation for message channel objects and their operations, since such a default translation would restrict the Generator to a single interpretation. Also, the Promela specification does not contain enough information to allow the translation to be completely automated. A translation will have to be defined manually for each communication construct as well as the operations on them. The Generator uses these definitions to generate correct models and implementations.

It is conceptually possible to translate Promela channels to communication objects, such as System V message queues and sockets, and to introduce additional extensions recognised by the Generator to translate communication operations. One obstacle, though, is that the Promela channel type is allowed as a formal parameter for processes. In an actual implementation it is usually necessary to generate additional information about the message structure associated with the communication object. Unlike systems like Spin and Löffler's implementation [84], where a special runtime environment exists to handle this, there is no default way to obtain the data associated with a specific communication object from just its

Promela	C	Range
bit	unsigned char	$0 \dots 255$
bool	unsigned char	$0 \dots 255$
byte	unsigned char	$0 \dots 255$
short	short int	$-2^{15} - 1 \dots 2^{1}5 - 1$
int	int	$-2^{31} - 1 \dots 2^{31} - 1$

Figure 8: Translation of the basic data types.

integer descriptor when it is declared in one process and passed on to another independently executing, and possibly distributed, process. It would be possible to generate additional code for specific cases, but would most likely be less efficient than an optimised implementation. Another approach would be to restrict the use of the PROMELA channel type as a formal parameter. This issue is not explored any further here, but may warrant further investigation.

3.1.6 Basic Data Types

PROMELA data types are translated to the nearest C representation. This means that a PROMELA data type will be represented by a C data type that has the same, or smallest data range just greater than that of the PROMELA version. In cases where the implementation version has a greater data range than the model version, truncation effects will not be the same, so users should be aware that this could cause differences in behaviour between model and implementation. It is also important to note that SPIN also stores an array of `bit`, `bool`, or `unsigned` variables internally as an array of byte variables, so it is generally not good practise to rely on wraparound or truncation effects.

The table in Figure 8 shows the implementation version for each PROMELA basic data type, except the `unsigned` type. The `unsigned` type is not equivalent to the unsigned type modifier of C, and does not have a default translation. The rightmost column shows the typical data ranges for the implemented types, but the actual data ranges are of course system dependent.

Arrays

PROMELA arrays are one-dimensional, but multi-dimensional arrays can be defined indirectly by using the `typedef` construct. For the implementation the base type of the array is simply translated as defined for that specific base type, and the array as a C array.

3.1.7 User-defined Types

```
TYPEDEF name '{' decl_lst '}'
```

Promela provides the `typedef` declaration to introduce user-defined data types. These user-defined types can be used anywhere predefined integer data types can be used.

Implementation

For every Promela `typedef` in the model a corresponding C type definition is generated: `typedef struct name {} name;`. These are declared in the global `init.h` header file so that all generated programs have access to the type.

3.1.8 Control Flow

Control flow refers to the order in which the individual statements are executed. Promela makes no distinction between conditions and statements, and the execution of every statement is conditional on its executability. Every basic statement has associated with it an executability, and this forms the basic means of synchronisation. For instance, instead of writing a busy-wait loop:

```
while (a != b) skip /* wait for a == b */
```

one can achieve the same effect in Promela with the statement

```
(a == b)
```

A condition can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does.

From this basic level, control flow is extended by forming sequences of statements. A sequence is formed by concatenating statement using the Promela statement separators, which can be either '`;`' or '`->`'. By encasing a sequence within a pair of curly braces a code block is formed, but this is usually only used for the `unless` statement.

Since processes in a Promela operational model all execute asynchronously, control flow is further extended by the fact that the execution of these statement sequences are interleaved to form the global system execution. To restrict interleaving, PROMELA allows the specification of atomic sequences which are executed as indivisible units. This is a very effective method for reducing the complexity of the verification model. Two versions are provided, `atomic{...}` and `d_step{...}`. A `d_step` sequence differs from an `atomic` sequence in it does not allow unconditional jumps in and out of the sequence, non-determinism, or blocking statements.

A recent addition to PROMELA is the conditional expression:

```
‘(’ expr ‘->’ expr ‘:’ expr ‘)’
```

It corresponds to the C conditional expression, but has a different syntax in order to avoid parse conflicts. As indicated in the syntax declaration, the enclosing parentheses are mandatory. The semantics are the same as in C, which is to say that the expression has the value of the second expression if the first expression evaluates to true (non-zero), and the value of the third expression otherwise. This expression is simply translated directly to its C form for the implementation.

Implementation

Since blocking is not directly supported in C, and since we want to keep our code as portable as possible without resorting to system-specific functions, we make use of busy-waiting in the generated implementation. This may seem inefficient, but this use of conditions was intended for abstraction of system behaviour. These conditional expressions can still be used as a powerful abstraction tool by limiting their use to the specification of the system environment. In this way they are not generated as part of the system implementation. Even if the user decides to use conditional statements in the part of the system specification that gets generated the system behaviour will be similar to that of the specification. In general the use of busy wait loop is not good programming practise and the generator will warn the user if such a statement is to be generated.

Sequential execution of concatenated statements translate directly to the implementation, with the only difference being that in C `‘;’` is a statement terminator, not a separator. All PROMELA statements are generated with trailing `‘;’` to avoid compiler issues.

Contrary to PROMELA, code in C programs execute non-interleaved, so interleaving or concurrency issues are only relevant for global resources (which should be modelled as such in PROMELA). Execution scheduling is handled by the supporting operating system or runtime, but this will be covered as one of the possible execution interleavings checked in the model.

Any statement sequence that in the actual implementation is guaranteed to not be part of a critical section, can be defined within an `atomic` or `d_step` statement to minimise the model complexity.

3.1.9 Selection

```
IF '::' sequence [ '::' sequence ] * FI
```

The selection construct provides a convenient way to define selection, or choice, in the structure of the underlying automaton. The body of the construct consists of one or more option sequences, with a sequence consisting of concatenated declarations and statements. The start of an option sequence is indicated with a double-colon, and the first statement after the double-colon is the guard statement for that option. An option can only be selected for execution if its guard statement is executable. When an option sequence is then executed, the default point for execution to continue is the control state that follows the selection.

Implementation

A selection construct is only executable if one of the guards is executable, and if more than one guard is executable, one is selected non-deterministically. This behaviour of course does not have a direct translation to the implementation, and is handled as follows. The selection is implemented as a C `if()` statement, with the guard statements as the option conditions and the rest of the statements of the option sequence as the body of the particular `if` option. The implementation, however, is not supposed to be non-deterministic, so an option is chosen deterministically, testing from top to bottom until an executable option is found. To handle the fact that a selection blocks until a guard becomes executable, the `if` statement is enclosed within a `while (1)` loop, and statements are added to each option so that it exits the loop after it has executed. An example of the translation of a basic selection statement is shown in Figure 9.

The else statement

If we where to know that a selection will always have an executable option, the loop would be unnecessary. Testing for this can be quite complicated, so the current version of the Generator only handles the special case where an `else` statement is specified. In Promela

```
if
:: (x == 1) ->
    printf("x == 1\n");
:: (y == 2) ->
    printf("y == 2\n");
fi
```

```
while (1) {
    if (x == 1) {
        printf("x == 1\n");
        break;
    }
    else if (y == 2) {
        printf("y == 2\n");
        break;
    }
}
```

Figure 9: Translation of a basic selection statement.

```
if
:: (x == 1) ->
    printf("x == 1\n");
:: (y == 2) ->
    printf("y == 2\n");
:: else ->
    printf("no match");
fi
```

```
if (x == 1) {
    printf("x == 1\n");
}
else if (y == 2) {
    printf("y == 2\n");
}
else {
    printf("no match");
}
```

Figure 10: Translation of the `else` statement.

`else` is a predefined condition statement that is only executable when none of the other options in the selection is executable. The `else` statement can be used as a guard for any option, and is translated with an `else` statement as part of the `if` statement. The translation is illustrated in Figure 10. As the example clearly shows, the enclosing loop is unnecessary, and it is therefore not generated.

The switch special case

The GENERATOR also tests for the special case where the selection can be implemented as a C `switch` statement, which is generally more efficient than an `if` statement. The criteria for a switch statement is that only one variable is tested and that it is of an integral type. We also limit it to cases where the variable is compared to a constant expression. An example is provided in Figure 11.

For this special case, a label is inserted right after the enclosing `while` statement, and `goto`

```
if
:: (x == 1) ->
    printf("x == 1\n");
:: (x == 2) ->
    printf("x == 2\n");
fi
```

```
while (1) {
    switch (x) {
    case 1:
        printf("x == 1\n");
        goto end_selection_1;
    case 2:
        printf("x == 2\n");
        goto end_selection_1;
    }
}
end_selection_1;
```

Figure 11: The translation of a selection to a C switch statement.

```
if
:: (x == 1) ->
    printf("x == 1\n");
:: (x == 2) ->
    printf("x == 2\n");
:: else ->
    printf("no match");
fi
```

```
switch (x) {
case 1:
    printf("x == 1\n");
    break;
case 2:
    printf("x == 2\n");
    break;
default:
    printf("no match");
}
```

Figure 12: A selection containing an `else`, translated to a C switch statement.

statements are used to transfer control to a label at the end of the loop. The reason for this is that a `break` statement will only exit from the `switch` statement, and not the `while` loop. Selection constructs are labelled numerically in the order in which they are specified or declared. The label `end_selection_n` will for example follow the nth selection construct. Note also that in this case it is not necessary to insert `break` statements at the end of each case option, since we already have the unconditional jump to avoid fall-through.

What if a selection construct satisfies the constraint for a switch statement and also has an else statement guard? In this case the switch statement will have a `default` option, and the enclosing loop is no longer necessary. In this case, the `goto` statements can once again revert to `break` statements, as shown in in Figure 12.

```
do
:: (x == 1) ->
    printf("x == 1\n");
:: (y == 2) ->
    printf("y == 2\n");
od
```

```
while (1) {
    if (x == 1) {
        printf("x == 1\n");
    else if (y == 2) {
        printf("y == 2\n");
}
```

Figure 13: Translation of a simple repetition statement.

3.1.10 Repetition

```
DO '::' sequence [ '::' sequence ] * OD
```

The repetition construct is similar to the selection construct in all ways but one; the end of each option sequence transfers control back to the start of the repetition construct. This allows for repeated execution. To exit a repetition construct the Promela `break` statement has to be used.

Implementation

The repetition construct is implemented much the same as the selection construct, only now the enclosing loop is mandatory to ensure repetition. Code to exit the loop is also now only added where the Promela `break` statement is specified. The specific code generated to handle the exit out of the loop depends on the specific translation. The translation of a basic repetition statement is illustrated in Figure 13.

For this simple example a Promela `break` statement can simply be replaced by a C break statement, but not all cases are as simple. The Promela `break` statement is specified to exit the innermost repetition construct. If such a repetition construct then contains any selection construct that is implemented using an enclosing loop or switch statement, a simple C break statement will not exit the correct construct. For these cases it is necessary to use a jump statement that exits to the label specified at the end of the innermost repetition construct, similar to what was done for the selection construct using a switch statement.

Figure 14 illustrates one such an example. The Generator needs to keep track of the innermost repetition construct, as well as the specific statements used to implement selection constructs in cases where they are combined so that the correct exit statement is generated. In this relatively simple example we can already see where `break` statements were generated to

```
1   do
2   :: (i == 0) ->
3       if
4       :: (j == 0) ->
5           printf("(0, 0)");
6       :: (j == 1) ->
7           printf("(0, 1)");
8       :: else ->
9           printf("invalid");
10          break;
11      fi
12  od
```

```
1   while (1) { /* selection #1 */
2   if (i == 0) {
3       switch (j) {
4       case 1:
5           printf("(0, 0)");
6           break
7       case 2:
8           printf("(0, 1)");
9           break
10      default:
11          printf("invalid");
12          goto selection1_end;
13      }
14  }
15  }
16  selection1_end:;
```

Figure 14: Control flow combining repetition and selection.

exit the `switch` (lines six and nine), and a `goto` jump statement to implement the PROMELA `break` statement specified in the last option of the inner selection construct (line 12).

3.1.11 Predefined Constants, Variables and Functions

For the most part, the predefined variables and functions in PROMELA are specific to modelling and few are used in a context directly relevant to the actual implementation. To put it another way, even if they model actual system objects and behaviour, they are generally used as an abstract representation. An example of this is the `timeout` predefined variable, which provides a mechanism to model a system timeout, but cannot be directly translated since it depends on the state of the complete system, something that cannot generally be known in an actual distributed system.

The predefined Boolean constants `true` and `false` are synonyms for the constant values one (`1`) and zero (`0`) respectively, and support a more natural syntax for manipulating Boolean values. The keyword `skip` is also translated to the constant value one (`1`), and functions as a dummy statement which is always executable and has no effect. Such a dummy statement is typically used to sidestep the PROMELA grammar rule that a statement sequence should contain at least one statement. Since `skip` is used as a statement, it translates to the statement `1;` in C, which is useless but still correct. Even more accurate would be to translate `skip` to the real empty statement in C, `;`, but since this does not make any difference in the final implementation it is not worth testing for this special case when generating the implementation.

The Promela functions `printf` and `assert` are translated to their C counterparts. The corresponding C libraries have to be included in the implementation, which is handled by inserting the corresponding C preprocessor directives into the `init.h` header file. For non-standard functions, such as the `run` operator (see Section 3.1.4), libraries were implemented. These are also included into the implementation through the `init.h` header file.

3.1.12 Embedded C Code

From version 4.0 onwards, Spin supports the inclusion of ANSI C code fragments in Promela through five new primitives. This mechanism provides support for automatic model extraction from C code, such as with FeaVer [68]. However, our approach is able to take full advantage of this functionality.

There are five keywords that introduce the C fragments:

```
c_code, c_decl, c_state, c_track, c_expr,
```

with the following syntax:

```
C_CODE [ ‘[’ c_expression ’]’ ] ‘{’ c_code ‘}’
C_DECL ‘{’ c_declarations ‘}’
C_STATE string string [ string ]
C_TRACK string string
C_EXPR [ ‘[’ c_expression ‘]’ ]‘{’ c_declarations ‘}’
```

The `c_code` primitive introduces embedded C code fragments, that are only executed if the optional C expression is true. If it is not present, it defaults to the expression `1`. The `c_decl`, `c_state`, and `c_track` primitives introduce C declarations. A `c_expr` can be used for guard conditions that are not expressible in Promela because of its more restrictive data types and language constructs.

The use of these primitives are not as straightforward as they might appear. Since they modify the behaviour of the model checking program generated by Spin, they can, in some cases, have subtle side-effects and must be used with care. A full discussion of these primitives and their use is beyond the scope of this thesis; as before, interested readers are referred to [65].

Implementation

The Generator inserts the content of embedded C code fragments directly into the implementation. Like Spin, the Generator does not parse the code fragments. However, any errors in the C code fragments will be reported by the C compiler when the implementation is compiled.

3.2 Extending Promela

We have now seen how most of the Promela constructs are translated, but for others there is simply no natural default translation. For these the translation is implementation specific. The dilemma we face is that in order to automatically generate a complete and efficient implementation the system has to be specified with enough detail, but for model checking we want to only specify the system behaviour relevant to the requirements to be verified to keep the model tractable. To address this problem the Generator accepts an extended version of Promela that allows the user to provide it with sufficient detail so that it can automatically generate an executable implementation. The Generator also produces a version of the specification in which all non-standard Promela language features are stripped away, making it completely compatible with Spin.

3.2.1 Embedded Code Fragments

In order to allow the specification of a complete implementation, the Generator extends Promela with a number of new primitives. These primitives are similar to the embedded C code primitives already provided by Promela, but are interpreted exclusively by the Generator. They may appear simple, but provide an elegant and powerful way to specify implementation detail not relevant to the verification, as well as define conversions and abstraction by specifying implementation and model specific code pairs, much like the tabled abstraction approach used by FeaVer, only in reverse. For these primitives we use the shorthands IS and MS to indicate *implementation specific* and *model specific* respectively, where implementation specific code becomes part of the implementation but not the final verification model, and model specific code is kept as part of the verification model, but does not become part of the implementation.

Implementation specific code blocks are introduced with the keywords:

```
c_code_IS, c_expr_IS, c_decl_IS,
```

and have the following syntax:

```
C_CODE_IS [ '[' c_expression ']' ] '{' c_code '}'
C_EXPR_IS [ '[' c_expression ']' ]'{' c_declarations '}'
C_DECL_IS '{' c_declarations '}'
```

These primitives can be used in exactly the same way as their PROMELA cousins, with the exception that they do not become part of the verification code. Another small difference is that the `c_decl_IS` statement can be used inside a PROMELA `typedef`, whereas the `c_decl_IS` statement cannot. This allows the definition of structured types with specific fields only present in the implementation, and not the model.

Model specific code blocks are introduced with the keyword:

```
promela_MS,
```

and has the following syntax:

```
PROMELA_MS '{' sequence '}'
```

The functionality provided by this primitive is the reverse of that provided by the primitives for embedded C code. The body of a `promela_MS` is standard PROMELA code, and must be syntactically correct within its definition scope. Contrary to the IS primitives, the GENERATOR does parse the body of the MS primitive. The PROMELA code contained within an MS statement is preserved in the model (without the wrapper), but is not generated as part of the implementation.

By combining the use of IS and MS statements, it is possible to formally define all kinds of PROMELA-to-C conversions and abstractions. For example, it is possible to specify:

- Code that is not relevant to the verification, but is needed to generate a complete implementation. This code is simply specified using the IS primitives.
- Abstraction. By specifying the implementation version of the code using an IS primitive, and a corresponding abstract version that better suits the verification using a MS primitive, abstraction is effectively applied. Any number of different kinds of abstraction is possible. One important example is the introduction of non-determinism for the

```
env proctype rand(chan res)
{
    if
    :: res!0
    :: res!1
    fi
}

#define RAND(var) \
promela_MS { \
    run rand(res); \
    res?var; \
}; \
c_code_IS{var = rand()%4;};

active proctype A()
{
    int x;
    promela_MS {
    chan res = [1] of {int};
    };
    do
    :: true ->
        RAND(x);
        printf("%d\n",x);
    od
}
```

```
proctype random(chan res)
{
    if
    :: res!0
    :: res!1
    fi
}

active proctype A()
{
    int x;
    chan res = [1] of {int};
    do
    :: true ->
        run rand(res);
        res?x;
        printf("%d\n", x);
    od
}
```

```
#include "init.h"

int main()
{
    /* Declarations: */

    int x = 0;

    /* Statements: */

    while (1) {
        x = rand() % 4;
        printf("%d\n", x);
    }
    selection1_end:;

    return EXIT_SUCCESS;
}
```

Figure 15: Abstraction of a function call.

purpose of verification; it sometimes suffices to use non-deterministic choice to model all possible outcomes of an operation, rather than include the specific details of the operation.

- Syntactical conversion. Similar to abstraction, model and implementation specific representations are specified for statements where differences in syntax between PROMELA and C do not allow a direct translation. Contrary to abstraction both versions of the statements provide the same functionality, just with different syntax.

To illustrate the use of this technique, Figure 15 shows an example of abstracting a function call. The leftmost column shows the complete specification, the centre column the corresponding PROMELA only file, and the rightmost column the generated implementation source for `proctype` A.

3.2.2 Additional Options

The only option currently provided by the GENERATOR is `env`, a special prefix that can optionally be specified before a `proctype` declaration and which identifies the process as part of the system environment. The syntax is as follows:

```
[ ENV ] proctype
```

To verify a system it has to be closed, which means that the model has to include the complete system with the environment it interacts with. Environment processes are, however, not intended to be part of the implementation. The `env` prefix simply informs the GENERATOR not to generate the process as part of the implementation.

3.3 The Generator

The GENERATOR resembles a compiler, but instead of compiling a high level language to machine code, it parses the extended PROMELA specification and generates an implementation in the form of ANSI C source code. The GENERATOR also generates a "clean" version of the specification that consists of only standard PROMELA, which is compatible with the SPIN verification system. The current version of the GENERATOR is a command line program, used as follows:

```
usage: generator [options] <filename.pch>
options:
        -c              : do not discard comments.
        -d <level>      : specify level of debugging output,
                          valid argument = [off,low,medium,high].
        -t              : tidy up the Promela output.
```

It takes as argument the name of the file that contains the complete specification. The default extension for this source file is **.pch**, which is an acronym for **Promela/C Hybrid**. The PROMELA only version of the specification is generated in a file with the same name as the source file, and the file extension changed to **.prom** (the default extension for PROMELA files). If the source file does not have the default **.pch** extension, the **.prom** extension is simply appended to the end of the filename.

The GENERATOR also accepts three optional arguments. The `-c` option indicates that the comments in the source file (.pch) should be preserved in the PROMELA only file (.prom).

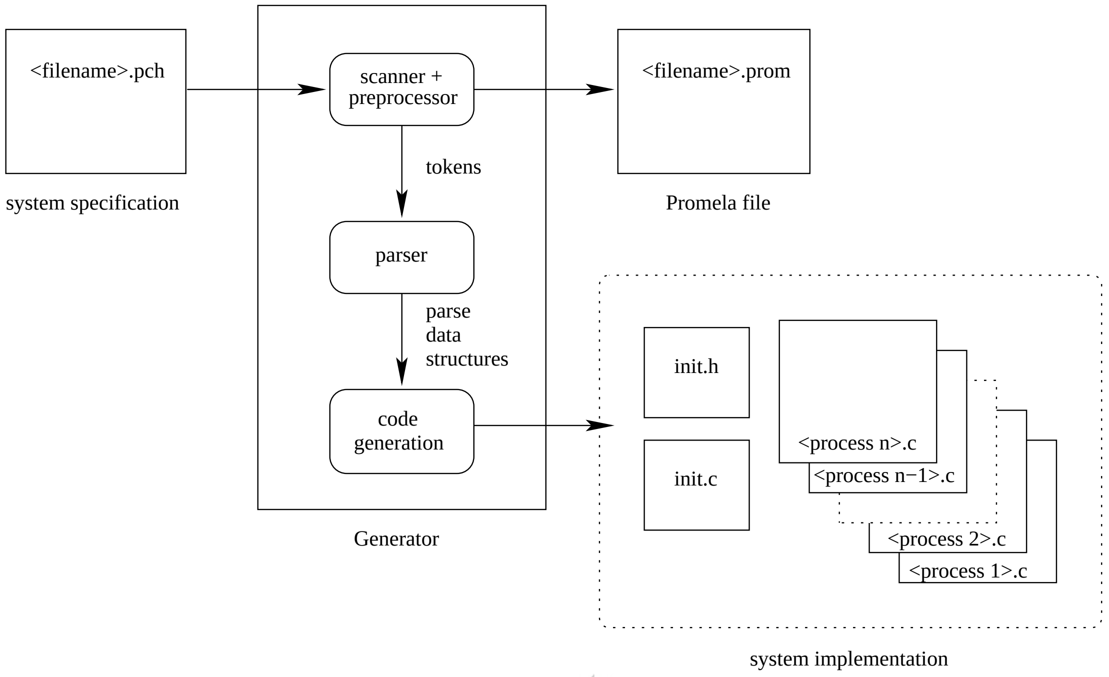

Figure 16: Generator component layout.

The `-d` option is used to specify the verbosity of the GENERATOR during executing, and is primarily used for debugging. Lastly, the `-t` option produces a tidier PROMELA model. When the generator removes the non-PROMELA elements from the source specification, it leaves the empty lines so that the line numbers match up with that of the original source file. If the `-t` option is specified, the GENERATOR processes the input, writes the `<filename>.prom` file, and generates a new file named `<filename>_tidy.prom`, in which successive empty lines are replaced with a single empty line.

3.3.1 Architecture

The main components of the GENERATOR are a scanner, parser and code generator. The component layout is depicted in Figure 16. There is of course also the supporting elements, which consist of the definition of the data structures and the functions that operate on them, as well as helper functions for error messages and other output.

The parser contains the main control function, and drives the whole system. When execution starts, the GENERATOR first processes all options, and then tries to open the file specified as argument. If the file exists, the GENERATOR is initialised. The source file is then preprocessed by **cpp**. This happens externally to the GENERATOR, and is invisible to the user. The preprocessed file is saved in a file named `preproc_<filename>`, where `<filename>` is the name of the source file. This preprocessed file is then used as the input for the scanner. The GENERATOR refers to the preprocessed file in all parse error messages to ensure that the line positions match up correctly.

3.3.2 Scanner

The current version of Spin uses a custom scanner, but the **flex** lexical analyser generator was used to implement a scanner for the GENERATOR. This scanner not only tokenizes the input file, but also handles the preprocessing that is not handled by the C preprocessor. While scanning the source file, the scanner also generates a PROMELA-only version of the specification. This is done on-the-fly by echoing the standard PROMELA elements from the source specification to the PROMELA-only specification.

The preprocessing that the scanner has to handle includes ANSI C comments, embedded code fragments and inline named sequences. The source file is first preprocessed by the **cpp** C preprocessor, which inserts linemarkers into the file for bookkeeping. The scanner has to match and remove these as well.

3.3.3 Parser

The parser parses the file processed by the scanner and builds data structures that contain all data needed to generate an implementation. After the whole specification has successfully been parsed, the code generation functions are called to generate an implementation. To ensure that the GENERATOR parses the exact same language as SPIN, the **Bison** grammar file from the most current version of the SPIN source (version 4.2.7) is used for the GENERATOR parser. The productions in the grammar file are not modified, but the actions replaced with code relevant to the GENERATOR.

If the parser encounters any errors while parsing, the user is notified, but parsing continues. When the parse process is finished and there are errors, the GENERATOR notifies the user and exits. Apart from errors there are also warnings produced by the GENERATOR for issues that are not fatal to code generation, but that the user should be aware of.

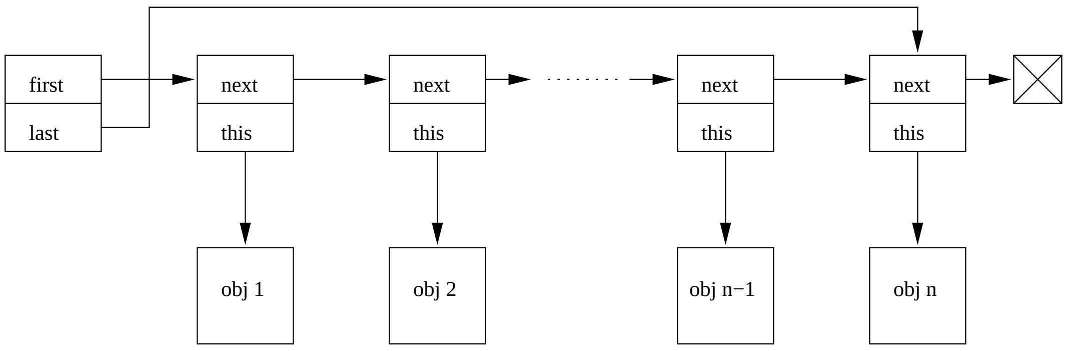

Figure 17: Linked list data structure.

3.3.4 Data Structures

The generator uses very basic data structures, namely trees and linked lists. During parsing a data object is generated for every basic element. The Generator keeps a linked list for every top level object. These linked lists all have the same general structure, as shown in Figure 17.

The linked list object has a pointer to the first and last element in the list to make insertion of list items easier and faster. A simple linear search is used to find elements in a list. A more complicated implementation is not warranted, since the lists contain relatively few elements, and such an optimisation will not produce any perceived improvement in performance.

The top level objects groups are :

- `proctype` declarations,
- global declarations and statements,
- `mtype` definition(s),
- user-defined types, and
- `inline` named sequences.

Processes

When parsing a `proctype` declaration, a new process data structure is created and inserted into the list of system processes. The data structure for processes is shown in Figure 18.

```
typedef struct proc
{
    int           env;           /* environment flag */
    struct active *active;       /* initial instantiation */
    struct node   *enabler;      /* optional process enabler */
    char          *name;         /* process name */
    struct seq    *prmtrs;       /* process parameters list */
    struct seq    *declarations; /* process body - declarations */
    struct seq    *statements;   /* process body - statements */
    int           lineno;        /* line number of proctype decl */
    int           linepos;       /* line position of proctype decl */
    struct proc   *nxt;          /* linked list of processes */
} Process;
```

Figure 18: Process type data structure definition.

As noted before, PROMELA allows declarations and statements to be mixed in the body of a `proctype`, but behaves as if all declarations are at the beginning of the body, before any of the statements. The GENERATOR therefore maintains separate sequence structures for declarations and statements. A sequence structure is a linked list, as described above. During parsing, statements are concatenated and assigned to the process data structure once the complete body has been parsed. The declarations, however, are directly appended to the list of statements kept for the process as they are parsed. In addition to all the system processes, a special global process is added to the list of processes to store the global declarations.

Sequences

The way declarations and statements are stored in a list structure follows naturally from their description in the grammar of PROMELA. A sequence in PROMELA is built up of sequential steps, with a step being either a declaration or a statement. Sequences make up the bodies of `proctype`, never claim, event trace, and inline declarations. Sequences are also found in `atomic`, `d_step`, and code block declarations, as well as in the options in selection and repetition constructs.

Abstract Syntax Tree

The actual declarations and statements that make up the steps in the sequences are parsed and stored as abstract syntax trees. The `this` pointer of each step points to the root of the tree. For example, the simple statement `x = x + 1` is represented by the structure in Figure 19. The internal nodes of the tree represent operators, and the leaf nodes represent

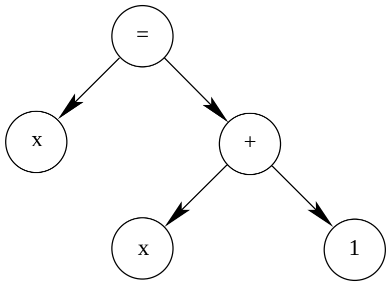

Figure 19: Sample syntax tree.

```
typedef struct node
{
    int          type;          /* node type */
    enum exec_tp executability; /* executability */
    char         *name;         /* named types */
    int          val;           /* variable value */
    struct seq   *seq;          /* optional sub sequence */
    int          lineno;        /* line number position */
    int          linepos;       /* line position */
    struct node  *ref;          /* parameter reference */
    struct node  *lft;          /* left child */
    struct node  *rgt;          /* right child */
} Node;
```

Figure 20: Tree node data structure definition.

the operands. Since the structure is later used to generate a high-level implementation, as much detail as is necessary is preserved. The C data structure for the nodes of the tree is shown in Figure 20.

3.3.5 Code Generation

After successfully parsing the source specification, the GENERATOR uses the data structures to generate the implementation in ANSI C. Code generation basically reverses the parsing process to re-implement the specification in a different implementation language. The data structures are traversed only once to generate the implementation. We discuss the steps of the generation process in the order they take place.

Global declarations

The global declarations are generated first, specifically because the `c_state` statement can possibly define variable declarations that are local to one of the processes. When these are encountered, the declaration is appended to the sequence of declarations of the specific `proctype` data structure. The declaration will then be correctly generated along with the rest of corresponding `proctype`.

The first step is to create the `init.h` header file. The preprocessor directives to include the necessary library functions are inserted, followed by the global declarations and statements. Possible global objects include the `mtype` enum declaration, user-defined types, global variable declarations, and global statements, and are generated in that order.

It is important to remember that each process will have its own private copy of these global variables; they are not global to the final application. The reasoning behind this limitation is that it is difficult to provide shared, global data in a distributed application without also providing a significant — and probably unwelcome — framework of support code. Such code may be added to the GENERATOR in the future, but it is unlikely to become the default mode of operation. The purpose of the GENERATOR is produce small, clean, and portable code for embedded systems. The GENERATOR therefore issues a warning for each global declaration. Nevertheless, global declarations do provide a clean way of sharing global constant between processes in the system (such as port numbers, etc.).

Processes

After the global objects have been processed, all of the `proctype` declarations are generated, each as a separate program implementation in ANSI C. The regular processes are generated first, so that the code for instantiating the `active` processes can be prepended to the statement sequence of the body of the `init` process. If no `init` process was declared in the specification, a data structure is created for one to handle the initialisation.

The processes are generated in the order in which they are declared in the specification, which means that their initialisation will also be added in that order to the `init` process. For each `proctype` declaration a `<name>.c` C source file is created, where `<name>` is the name of the process. The `init.h` file is included into every `proctype` implementation to share global types and constants. The C code implementing the body of the `proctype` is then generated in this source file, within a C `main()` function. The variables for the formal parameters are declared and initialised first, followed by the local variable declarations. Lastly the code implementing the statement sequence that make up the rest of the process body is generated, and the file closed.

init

After the implementations for all regular `proctypes` have been generated, the init process is generated. The implementation code for the body of the `init` process is generated in the `init.c` source file, with the `init.h` header file already created for global objects. The header file is of course also included into the `init.c` file.

Statements and Declarations

The generation of the statement sequence is reminiscent of a recursive descent parser; the objects that represent the parsed specification are passed top-down by ever refining functions until the function for the basic element is reached and an implementation is generated for that element. A special case is the generation of `inline` calls, which have to be processed first to insert the actual arguments before being generated. This can be tricky, since they can include other `inline` calls, and can also appear at the start of a selection option, which means that the guard statement has to be extracted from the inline code.

Generator Library Functions

The GENERATOR libraries that are included by default into the generated implementation provide commonly used functions. These include, for example, the `run` function, output functions, and Internet socket functions. These functions were implemented with built-in error handling. This approach cleans up the source specification. As long as it is guaranteed that the PROMELA version for each function satisfies the same pre- and postconditions, the implementation detail contained within these functions can be ignored (for the model).

The GENERATOR currently includes the following libraries:

- genliboutput : General output functions for error messages and debugging.
- genlibproc : Contains the `run()` function.
- genlibipc : System V IPC functions.

- genlibsocket : Internet socket functions.
- genlibftp : FTP functions used for the FTP server implementation.

These library functions are linked into every implementation by default. If any of these are not needed, the user can optimise the implementation by manually removing the corresponding library file name from the Makefile entry for the specific program. Fortunately, most modern linkers discard unused code, unless instructed otherwise.

Makefile

In addition to the C source code implementation, the GENERATOR also generates a Makefile for the specific implementation. It contains entries for each `proctype`/program, with all the required command line options. This makes it possible for the user to compile the complete generated implementation by simply issuing a `make` command at the console. Even if the user were to manually edit the generated source files, the Makefile should still work correctly (barring such drastic acts as changing the filenames, of course).

Chapter 4

Case Studies

In this chapter the use of the GENERATOR and the general technique will be illustrated by a couple of case studies. We will first look at an implementation of the alternating bit protocol to familiarise the reader with the basic approach, and then move on to an efficient implementation of an FTP server as an example of a non-trivial application.

4.1 The Alternating Bit Protocol

The alternating-bit protocol as first described by Bartlett et al. [4] was developed for reliable full-duplex transmission over half-duplex links. The method uses a singe control bit, called the alternating bit, to ensure the correct transmission of messages between a sender and receiver. Each message also contains error detection information, and the method can be shown to be infallible assuming all transmission errors are detected.

Figure 21 shows two finite-state automata that represent the procedure for two communicating terminals A and B. The automata states have been numbered for the purpose of the discussion. The edges of the automata are labelled: underlined labels indicate messages being transmitted and non-underlined labels messages indicate being received.

As is clearly visible in Figure 21, each automaton is symmetrical with respect to the other, but also with respect to itself. It is possible to express the protocol more compactly by "folding" each automaton in half, thus reducing the number of states by two. However, to keep the discussion simple, we will base our implementation on the original protocol description. Let us first look at how the protocol would be modelled in PROMELA.

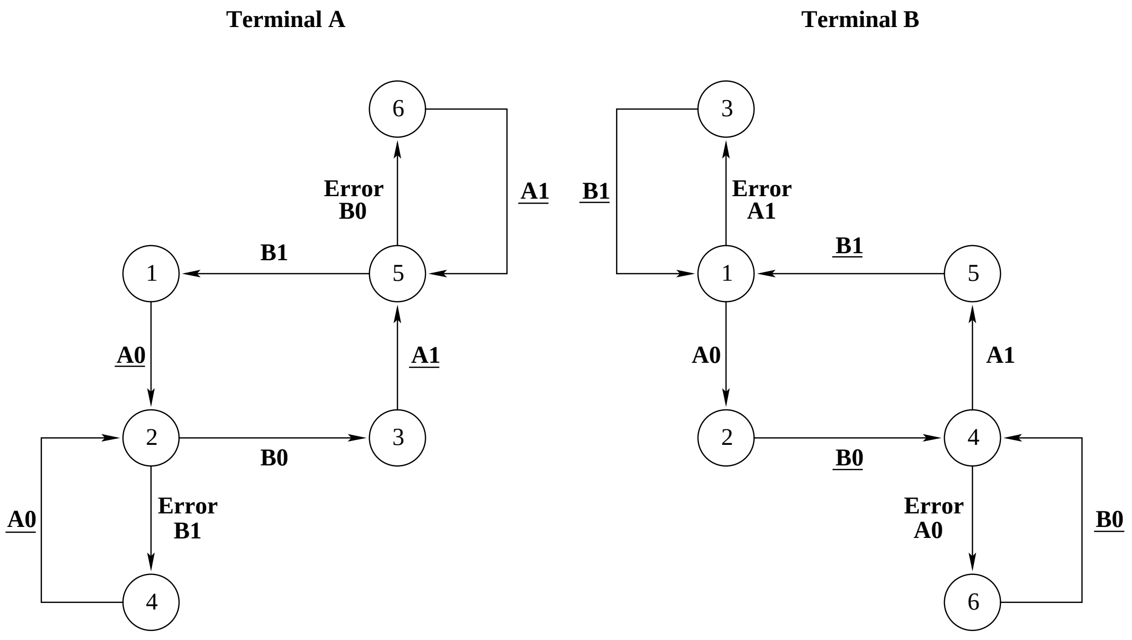

Figure 21: Finite-state automata representing the alternating-bit protocol.

4.1.1 Promela Specification

The terminals A and B are specified as the Promela processes A and B, and a simple first implementation is shown in Figure 22. To make the specification more readable a Promela `mtype` declaration is used to enumerate the states:

```
mtype = {STATE1, STATE2, STATE3, STATE4, STATE5, STATE6};
```

The transmission link is modelled using two Promela message queues, one for each direction of communication:

```
chan AtoB = [1] of { bit };    /* Sender to Receiver */
chan BtoA = [1] of { bit };    /* Receiver to Sender */
```

Here we can see that the messages contains the bare minimum, just the alternating-bit. Having the two processes communicate directly through the message queues means that the actual communication link is considered fault-free. This is of course not a very realistic and useful abstraction since we want to verify the correct operation of the protocol in the case of transmission errors. We'll address this issue shortly.

Using the `init` process to initialise the system, we can do a simulation run using XSpin for a quick inspection of the system behaviour. Figure 23 shows the message sequence chart generated during the simulation run, and the model appears to behave as expected.

```
proctype A(chan in, out) {
    mtype state = STATE1;
    do
    :: state == STATE1 ->
        out!0; state = STATE2
    :: state == STATE2 ->
        if
        :: in?0 -> state = STATE3;
        :: in?1 -> state = STATE4;
        fi
    :: state == STATE3 ->
        out!1; state = STATE5;
    :: state == STATE4 ->
        out!0; state = STATE2;
    :: state == STATE5 ->
        if
        :: in?1 -> state = STATE1;
        :: in?0 -> state = STATE6;
        fi;
    :: state == STATE6 ->
        out!1; state = STATE5;
    od
}
```

```
proctype B(chan in, out) {
    mtype state = STATE1;
    do
    :: state == STATE1 ->
        if
        :: in?0 -> state = STATE2;
        :: in?1 -> state = STATE3;
        fi;
    :: state == STATE2 ->
        out!0; state = STATE4;
    :: state == STATE3 ->
        out!1; state = STATE1;
    :: state == STATE4 ->
        if
        :: in?1 -> state = STATE5;
        :: in?0 -> state = STATE6;
        fi;
    :: state == STATE5 ->
        out!1; state = STATE1;
    :: state == STATE6 ->
        out!0; state = STATE4
    od
}
```

Figure 22: Promela proctypes for terminal A and B.

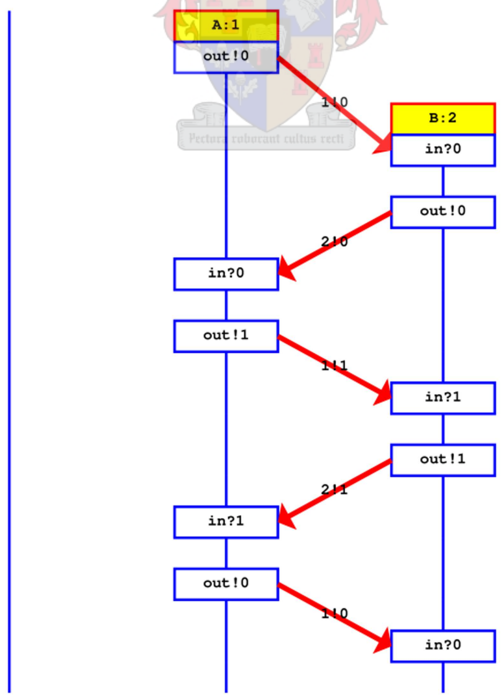

Figure 23: Message sequence chart for the basic alternating-bit model.

The chart shows the execution trail of the system corresponding to one cycle of the automata, with the last step again indicating the initial system state where terminal A again sends the message with the alternating-bit set to zero. The `init` process is already instantiated at the initial system state (hence the name) and is represented by the leftmost vertical timeline of the chart. The next two timelines show the instantiation of processes A and B, which are assigned process id's 1 and 2, respectively. The blocks indicate communication events, with the arrows indicating the direction of communication. The arrow labels show the numeric identifier of the specific message queue, followed by the bang sign (`!`) and the actual value of the message sent.

Terminal A sends the first message with the alternating-bit set to zero. Terminal B receives this message and acknowledges with a message containing an alternating-bit also set to zero, thus acting as an "acknowledgement" bit. Both terminals "alternate" their alternating-bits upon a successful transmission, and the behaviour is repeated.

Having described the basic model, let us take a look at how to specify an executable implementation, and then introduce a more realistic environment abstraction.

4.1.2 Specifying an Implementation

The specification for the implementation builds on the model just discussed. The control structure is exactly as shown for the model, completely specified in PROMELA. The extensions to PROMELA, as described in the previous chapter, are used where necessary to bridge the differences in syntax and semantics, or abstraction is needed for a tractable model.

As discussed in Subsection 3.1.5, the GENERATOR provides no default translation of message passing. For this particular application it was decided to use System V message queues, since they closely resemble PROMELA channels. This simplifies the specification to such a degree that we can focus on the technique and not on irrelevant technical details. Message passing is specified as follows:

```
promela_MS{
chan AtoB = [1] of { Msg };    /* Sender to Receiver */
chan BtoA = [1] of { Ack };    /* Receiver to Sender */
};
c_decl_IS{
int AtoB = _ipc_mq_new();      /* Sender to Receiver */
int BtoA = _ipc_mq_new();      /* Receiver to Sender */
};
```

The code contained in the `promela_MS` code block declares the channels for the model and replaces the previous declarations of `AtoB` and `BtoA`, while the code in the `c_decl_IS` declares the message queues with the same names for the implementation. The names are used throughout the specification and refer to the appropriate objects for both the model and implementation. Promela channels are uniquely identified by an integer value, so the `chan` type can be translated to an integer in C. This allows the use of implementation communication objects that are uniquely identified by integer descriptors, such as System V message queues and sockets.

The `_ipc_mq_new()` function call in the C code segment is a wrapper function that calls the System V function to get a descriptor for a new message queue, and handles all possible system errors. Verifying that such a function correctly corresponds to the Promela code it implements can be done independently from the rest of the system. The function is provided by the Generator libraries which are included in the `init.h` header file.

Whereas the messages in the first example only contained the alternating-bit, more realistic message structures are defined for the implementation specification:

```
typedef Msg {                           typedef Ack {
    c_decl_IS{long type;};                  c_decl_IS{long type;};
    bit     altbit;                         bit     altbit
    byte    data;                       };
    byte    checksum
};
```

In contrast to the finite-state automata representing the communicating terminals A and B, we only consider data transfer in one direction, from A to B — hence the separate acknowledgement message data structure. System V message queues require a message type identifier as the first field of the message, which explains the implementation-only `long type` declaration. It is interesting to note that this message type can be used in much the same way as in Promela to receive a specific message from the queue, by providing a constant argument for the first field in a receive that corresponds to the first value of the message we want to receive. The `Msg` and `Ack` message are relatively small. When dealing with larger messages it may be prudent to declare some of the fields inside a `c_decl_IS` construct, so as not to clutter the model specification unnecessarily.

The send and receive commands are then defined using simple C macros:

```
do
:: state == STATE1 ->
    msg.altbit = 0; msg.data++;
    SEND(out,msg,sizeof(Msg))
    state = STATE2
:: state == STATE2 ->
    RECV(in,ack,sizeof(Ack))
    if
    :: ack.altbit == 0 -> state = STATE3
    :: ack.altbit == 1 -> state = STATE4
    fi
:: state == STATE3 ->
    msg.altbit = 1; msg.data++;
    SEND(out,msg,sizeof(Msg))
    state = STATE5;
:: state == STATE4 ->
    SEND(out,msg,sizeof(Msg))
    state = STATE2;
```

```
while (1) { /* selection #1 */
if (state == STATE1) {
    msg.altbit = 0; msg.data++;
    _ipc_mq_snd(out,&msg,sizeof(Msg),1);
    state = STATE2; }
else if (state == STATE2) {
    _ipc_mq_rcv(in,&ack,sizeof(Ack),0,1);
    switch (ack.altbit) {
    case 0: state = STATE3; break;
    case 1: state = STATE4; break;
    } }
else if (state == STATE3) {
    msg.altbit = 1; msg.data++;
    _ipc_mq_snd(out,&msg,sizeof(Msg),1);
    state = STATE5; }
else if (state == STATE4) {
    _ipc_mq_snd(out,&msg,sizeof(Msg),1);
    state = STATE2; }
```

Figure 24: Specification and corresponding implementation for part of terminal A.

```
#define SEND(CHAN,MSG,SIZE)\
promela_MS{CHAN!MSG;};\
c_code_IS{_ipc_mq_snd(CHAN,&MSG,SIZE,1);};

#define RECV(CHAN,MSG,SIZE)\
promela_MS{CHAN?MSG;};\
c_code_IS{_ipc_mq_rcv(CHAN,&MSG,SIZE,0,1);};
```

where `_ipc_mq_snd()` and `_ipc_mq_rcv()` again are wrapper function for the corresponding System V message queue send and receive functions, and are also part of the libraries provided by the Generator. The behaviour of Promela send and receive statements depends on their location in the model. These statements may block depending on whether the channel they operate on is either full or empty, but guard statements should not block. The wrapper functions for the implementation incorporate this by providing a parameter that indicates whether it should block or just fail if a send or receive is not possible.

The bodies of the proctype declarations are updated to incorporate these new changes, resulting in a complete, implementable specification. The left column in Figure 24 shows a section of the specification for terminal A, corresponding to the first four states of the automaton (i.e., the first half). The rest of the specification follows the same pattern. The column on the right shows the corresponding generated implementation. The formatting of the implementation code has been edited so that the specification and implementation code line up

```
$ ./init
          [sender]         [receiver]

state1, sent msg = {0, 1}

                  state1, received msg = {0, 1}

                  state2, sent ack = {0}

state2, received ack = {0}

state3, sent msg = {1, 2}

                  state4, received msg = {1, 2}

                  state5, sent ack = {1}

state5, received ack = {1}
```

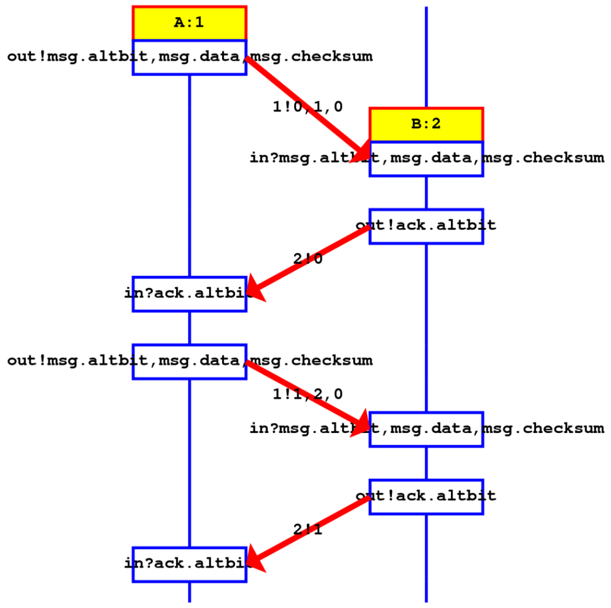
A:1
out!msg.altbit,msg.data,msg.checksum
1!0,1,0
B:2
in?msg.altbit,msg.data,msg.checksum
out!ack.altbit
2!0
in?ack.altbit
out!msg.altbit,msg.data,msg.checksum
1!1,2,0
in?msg.altbit,msg.data,msg.checksum
out!ack.altbit
2!1
in?ack.altbit

Figure 25: Implementation execution compared to model execution.

correctly. The layout of the actual code generated for the implementation is less terse and therefore more readable.

Now that the specification is complete, the next step is to generate the implementation. The Generator reads the file `altbit.pch` as input and generates a complete C source code implementation, as well as a Promela-only file `altbit.prom` and a Makefile to automatically compile the implementation. For this specific specification, the implementations for the processes `A`, `B` and `init` will be compiled to generate the programs `A`, `B` and `init`, respectively. The whole process from specification to implementation consists of two simple commands: `./generator altbit.pch` and `make`. Using Spin the file `altbit.prom` can be simulated and model checked to verify the control flow of the implementation.

The generated implementation can be executed by simply executing the `init` process. For illustration purposes, the specification was laced with output statements to show a sample execution. Figure 25 shows the effect of these statements against a message sequence chart produced by XSpin during a simulation run of the model. The model and implementation share the same control structure, so, as long as the actions for both satisfy the same pre- and postconditions, they exhibit exactly the same behaviour, as is clear from the figure.

```
env proctype link(chan fromA, toA, fromB, toB )
{
    Msg msg; Ack ack;
    :: fromA?msg ->
        if
        :: true -> toB!msg      /* correct transmission */
        :: true -> skip         /* lose the message */
        :: true ->              /* corrupt the message */
            msg.altbit = (msg.altbit+1)%2; toB!msg
        fi
    :: fromB?ack ->
        if
        :: true -> toA!ack      /* correct transmission */
        :: true -> skip         /* lose the ack */
        :: true ->              /* corrupt the ack */
            ack.altbit = (ack.altbit+1)%2; toA!ack
        fi
}
```

Figure 26: Communication link abstraction.

4.1.3 Verification

Sample executions provide a good indication of the system behaviour, but to ensure correctness we have to resort to verification. This first example is of course somewhat trivial compared to more realistic applications, but it illustrates some of the techniques available to enable efficient modelling and verification.

The original specification presents a very naive representation of the system environment, and does not allow for message loss or corruption. In other words, the sender, Terminal A, never visits states 4 and 6, and the receiver, Terminal B, similarly never visits states 3 and 6. The most natural way to introduce message loss is to specify a process that represents the communication link. The sender and receiver would then send and receive messages to and from the communication link. This is analogous to processes communicating via the physical link layer in an actual implementation. The communication link process would then use non-determinism to model the random possibility of lost messages or corruption of the message content. Such a process is shown in Figure 26.

The `env` prefix of the `proctype` declaration tells the GENERATOR that the process is part of the system environment and should not be generated as part of the implementation. Unfortunately, this approach leads to a steep increase in the number of states which, for larger models, renders the verification step infeasible.

One alternative to the communication link is to move the non-deterministic choice of whether or not to lose messages to the point where the messages are sent in process `A`. This mechanism can also model the situation where the message is not lost, but merely delayed. This is perceived by the receiver as a timeout, thus a premature timeout. The specification is modified by adding the following code:

```
CHAN?MSG;
if
:: true -> res = true;
:: true -> res = TIMEOUT;
fi;
```

The `res` variable is used to decide the next system state. In the implementation, the code is rendered as:

```
c_code_IS{res = _ipc_mq_rcv(CHAN,&MSG,SIZE,0,1);};
```

where the function `_ipc_mq_rcv()` is implemented to return the same values as its Promela counterpart. Only the sender process needs to be modified in this way. The implementation for the receiver process remains the same as before.

The same technique can be used to model the possibility of message corruption, with the actual implementation using a checksum function to set the value of `res` and the model using a non-deterministic choice to model the possibility of an error. Error detection is added to both the sender and receiver processes.

Once the system has been completely specified, it can be verified against correctness properties using Spin. One such a property is that `‘‘every message sent by A will be received error-free at least once and at most once by B’’`. The LTL version of this property is

$$\mathbf{G}\ (p \Rightarrow \mathbf{F}\ q)$$

where

$p \equiv$ “the message m has been sent by A”, and
$q \equiv$ “the message m has been received by B”.

If we define a variable `bit altbit` for both processes A and B, a possible LTL specification of this property would be (specified in the format understood by Spin):

```
[] ((s0 && X(s1)) -> (r0 && (<> r1)) )
```

with the atomic propositions defined as:

```
#define s0      (A:altbit == 0)
#define s1      (A:altbit == 1)
#define r0      (B:altbit == 0)
#define r1      (B:altbit == 1)
```

In the LTL property the fact that the alternating bit is flipped at the sender is seen as indicating a message send, and the fact that the alternating bit is flipped at the receiver as a message receive. The property then states that if a message is sent, it will eventually be received. We can use SPIN to verify that the model satisfies this property. For the case without the intermediate link (i.e., no communication errors) SPIN also indicates the parts of the code that are not visited. The attentive reader would have noticed that, even though it was said that SPIN does not support the next operator (**X**), it is used in this LTL formula. Support for the next operator has been added to the translation algorithm from LTL to Büchi, but this option is not enabled by default in the generated verifier. Unfortunately, SPIN's partial order reduction is incompatible with the **X** operator, and it is therefore usually disabled. In this case it was manually enabled, and partial order reduction was not used. Now that we have a basic background in the application of the technique, including the use of the GENERATOR, let us move on to the implementation of a more substantial application, an FTP server.

4.2 FTP Server

The File Transfer Protocol, or FTP, is a well known protocol for sharing files over a TCP/IP network. As an investigation into how well the proposed technique scaled, an FTP server was implemented. The protocol is an open standard, with the official specification readily available [100]. Also, FTP is a user-space protocol, and available on most modern systems. This makes it possible to easily compare our generated implementation with industry standard implementations.

FTP, like many communication protocols, is client-server based. The FTP server performs file related functions upon request of a user. An FTP session is instantiated by setting up a control connection and logging in to the server. The control connection follows the Telnet protocol [99], which means a user can connect directly to the server using a Telnet capable terminal, or indirectly using an FTP client program. Once connected, the client uses the

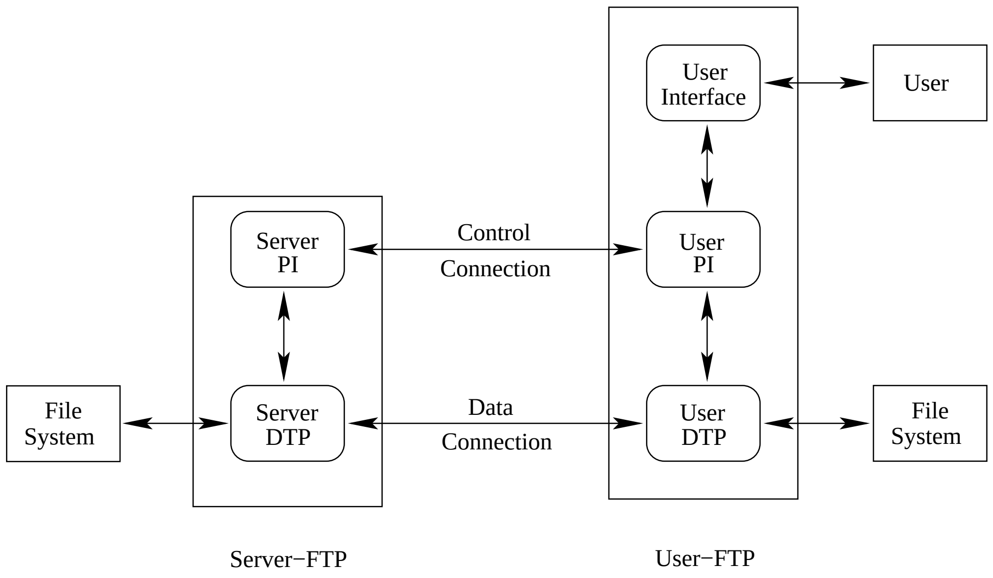

Figure 27: Model for FTP use (RFC 959, Figure 1).

control connection to initiate file manipulation operations such as downloading files from the server or altering the files located on the server. For the transfer of files a separate connection is set up, the data connection. The default behaviour is for the server to initialise the data connection between itself and the client, but the client can also set up data connections between servers. The first and most common situation is illustrated in Figure 27.

The client side of the protocol, called the user-FTP, includes a client specific protocol interpreter (PI), data transfer process (DTP) and a user interface. The user protocol interpreter, or user-PI, initiates the control connection from the port U to the server-FTP process, initiates FTP commands, and governs the user data process, or user-DTP, if it is part of the file transfer. The default behaviour of the user-DTP is to "listen" on the data port for a connection from a server-FTP process. If the data transfer is between two servers the user-DTP is inactive.

The server side of the protocol, called the server-FTP, consists of one or more processes which perform the function of file transfer in cooperation with a user-FTP and, possibly, another server. It includes a server specific protocol interpreter (PI) and data transfer process (DTP). The server protocol interpreter, or server-PI, "listens" on port L for a connection from a user-PI and establishes a control communication connection. It receives standard FTP commands from the user-PI, sends replies, and governs the server-DTP. The server data transfer process, or server-DTP, in its normal "active" state establishes the data connection with the "listening" data port. It sets up parameters for transfer and storage, and transfers data on command from its PI. The DTP can be placed in a “passive” state to listen for, rather than initiate a connection on the data port.

4.2.1 System Specification

To model, as well as implement the system, we first need to define the processes that will make up the system. Although we are only implementing the server side of the protocol, we still have to close the system for verification. This means that we will have to define processes for both the client and server. Since a server must be able to handle multiple connections at a time, it was decided to define a `ftpd` process that would listen for the connection from the user-PI, and then spawn a new `handler` process for each session. This isolates the majority of the control behaviour in the `handler` process, and frees up the part of the server that listens for control connections in order to ensure responsiveness. For the client side we only need a bare-bones, abstract implementation to drive the system. A process `clientPI` was implemented for this purpose.

The next step is to define the communication objects. Since the FTP protocol uses the TCP layer to communicate, Promela channels provide a good abstraction. A TCP connection endpoint is uniquely identified by an IP address and port number, and is referenced with an Internet socket descriptor. This means that for each unique connection type in the implementation a unique Promela channel is declared. One of the benefits of using Promela is that we can construct the system in parts and use Spin to look at the behaviour of the incomplete system, whereas implementing directly in C generally requires a lot more detail before we can inspect its behaviour.

Control Connection

The first part to be specified is the connection that is created when a client connects to the server. We first define the communication objects:

```
promela_MS {chan listen_sock = [0] of {chan};};
```

and

```
c_decl_IS {int listen_sock;};
```

where the PROMELA `listen_sock` channel is declared globally in the model so that the client processes can access it. In the implementation the clients use the known address of the server to connect to the socket, so it can be declared locally to the server process. On the accepting side of a connection, the `Accept()` socket operation returns a new socket that is the local endpoint for the new connection. This is modelled in PROMELA by sending the connection channel over the `listen_sock` channel. For the implementation it is necessary to initialise the `listen_sock` first:

```
c_code_IS {
    listen_sock = Socket(PF_INET, SOCK_STREAM, 0);
    liaddr.sin_family = AF_INET;
    liaddr.sin_port = htons(SERV_CNTRL_PORT);
    liaddr.sin_addr.s_addr = INADDR_ANY;
    Bind( listen_sock, (struct sockaddr *) &liaddr, addrlen);
    Listen(listen_sock, MAXQ);
};
```

This is an example of the use of the `c_code_IS` primitive to introduce implementation detail that does not need to be present in the model. With all initialisation taken care of, the `ftpd` process enters a non-terminating loop in which it waits for a connection from a client. After accepting a connection from the client, the `ftpd` process instantiates a `handler` process and passes it the control connection descriptor as an argument. This is specified as:

```
do
:: (true) ->
    promela_MS {listen_sock?control_sock;};
    c_code_IS {control_sock = Accept(listen_sock, &ciaddr, &addrlen);};
    run handler(control_sock);
od
```

This is a good example of how an IS and MS statement can be paired to define a conversion mapping for the C `accept()` statement to a PROMELA equivalent. Note that the `Accept()` function call in the example starts with a capital letter, and is thus not the standard `accept()` socket function. This is an example of one of the functions provided by the GENERATOR libraries. This specific function is part of a GENERATOR library for all socket related functions, where the actual function is wrapped in a wrapper function with the same name, but with the first letter capitalised. As described in the previous chapter, these functions has error handling built-in. This means that in the case of an error, the function will handle the error without crashing. This usually just means displaying an error message and returning a corresponding value if necessary. This cleans up the code and makes specifying the system easier.

Since the client process initiates the control connection, it also initialises the communication object:

```
promela_MS {chan control_sock = [0] of {int,int,int};};
c_code_IS {int control_sock = Socket( PF_INET, SOCK_STREAM, 0 );};
```

The implementation again contains initialisation information that is not relevant in the model, which we will not show here. The client sets up a control connection with the server by sending a connection request to the port the server is listening on, and then waits on the `control_sock` descriptor for a reply:

```
/* connect to server */
promela_MS {listen_sock!control_sock;};
c_code_IS {Connect(control_sock, &serv_addr, sizeof(serv_addr));};
/* wait for response from server */
promela_MS{control_sock?reply;};
c_code_IS {Recv(control_sock, reply_buff, sizeof(reply_buff), 0);};
```

As previously stated, the control connection follows the Telnet protocol, with the client issuing FTP service commands and the server replying with FTP reply codes. The commands are Telnet character strings that begin with a command code, followed by one or more spaces `<SP>`, followed by an argument field. The command codes are four or fewer alphabetic characters (case in-sensitive), and the argument field consists of a variable length character string ending with the character sequence `<CRLF>` (Carriage Return, Line Feed). An FTP reply is a string that consists of a 3-digit code (transmitted as three alphanumeric characters), followed by Space `<SP>`, followed by one line of text, and terminated by the Telnet end-of-line code `<CRLF>`. The code is used by the program to determine which state to enter next, whereas the text is intended for the human user. This means that the numeric codes have fixed definitions, but the information contained in the text is server-dependent and irrelevant to the program.

The `handler` process implements a state transition system. After receiving a command from the client the process uses the current state and the event (identified by the command) to execute the corresponding action. This control structure is implemented as select statements inside a non-terminating loop. The outside select switches on the state, and contains an inner select statement that switches on the event. Figure 28 shows a part of the code for the `handler` process as an example of the structure. The command receive statement is specified as follows:

```
promela_MS {control_sock?cmd,arg1,arg2;};
c_code_IS {cmd = get_command(control_sock, parameter);};
```

```
if
:: (state == AWAIT_USER) ->
    if
    :: (event == USER_CMD) ->
        REPLY(control_sock, 331, "User name okay, need password")
        state = AWAIT_PASS;
    :: (event == SYST_CMD) ->
        REPLY(control_sock, 215, "UNIX")
    :: (event == ERR_CMD) ->
        REPLY(control_sock, 530, "Please login with USER and PASS");
    :: else ->
        REPLY(control_sock, 501, "Command not implemented");
    fi
:: (state == AWAIT_PASS) ->
```

Figure 28: Part of the server code structure.

The `get_command()` is provided as one of the GENERATOR's library functions. Since PROMELA does not support strings, the integer value of the commands and replies are used directly. Reply statements are specified as follows:

```
promela_MS {control_sock!215,0,0;};
c_code_IS {send_reply(control_sock, "215 UNIX\r\n");};
```

In order to share the constant definitions of the FTP commands between the model and the implementation they are simply defined inside a header file, `ftp_constants.h`, which is included in both the model and the C code that implements the FTP functions.

To illustrate the behaviour of the system, a simulation of the initial setup procedure for an FTP session is shown in Figure 29. The leftmost timeline shows the `ftpd` process, since it is instantiated first. The `clientPI` process starts setting up the control connection by sending the connection request to the server. This request contains the channel id for the eventual control connection. Upon receiving this request the server spawns a `handler` process and passes it the control connection with id 1. The `handler` process notifies the client that the service is ready by sending it the reply `220`. The client then sends a `USER` command with the name of the user. The server recognises the name and replies with a request for the users password (reply code `331`). The server state now changes from `AWAIT_USER` to `AWAIT_PASS`. The client then sends the `PASS` command with the user password. (This is all in open text, one of the major drawbacks of standard FTP.) The server verifies the password, and if correct it sends the `230` reply to indicate success. The server state then changes to `LOGGED_IN`, and the client can perform file operations.

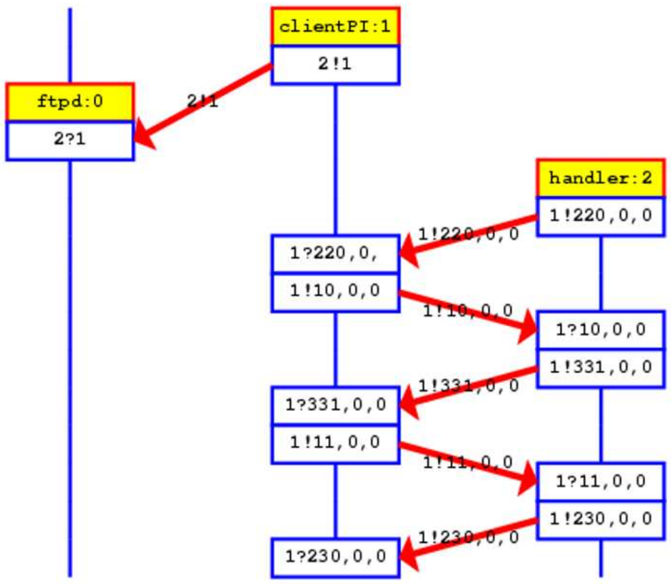

Figure 29: Session initialisation.

Data Connection

The mechanics of transferring data consists of setting up the data connection to the appropriate ports and choosing the parameters for transfer. Both the user and server-DTPs have a default data port. The client side default port is the same as the control connection port (U), and the server side port is adjacent to the control connection port (L-1). The transfer byte size is 8-bits.

The passive DTP "listens" on the data port prior to sending a transfer request command. The FTP request command determines the direction of the data transfer. The server, upon receiving the transfer request, will initiate the data connection to the port. When the connection is established, the data transfer begins between DTPs, and the server-PI sends a confirming reply to the user-PI.

The user also has the option to specify an alternate data port by using the `PORT` command. This can be used to set up a connection between two servers. The client commands one server to "listen" for a connection, and sends a `PORT` command to the other server indicating the data port of the first. Finally, both are sent the appropriate transfer commands.

The attributes of a data connection are:

- `TYPE` : The data representation type of the file to be transferred. This can be either

`ASCII`, `EBCDIC`, `IMAGE`, or `LOCAL` type.

- `FORMAT` : The format (used for ASCII or EBCDIC transfer). This is to indicate what kind of vertical format control is associated with a file.
- `MODE` : The transmission mode.
- `STRU` : The data structure. This can be either file, record, or page structure.

The FTP server was implemented to satisfy the minimum implementation requirements as defined in the RFC. These are:

- `TYPE` : ASCII Non-print
- `MODE` : Stream
- `STRUCTURE` : File, Record
- `COMMANDS` : USER, QUIT, PORT, TYPE, MODE, STRU, RETR, STOR, NOOP

The default values for transfer parameters are:

- `TYPE` : ASCII Non-print
- `MODE` : Stream
- `STRU` : File

4.2.2 Verification

Since the implementation and model share the exact same control structure, any verification results achieved for the model should apply to the implementation. More specifically, if it is shown that the model satisfies a property, the implementation will also satisfy this property. On the other hand, it is possible that a property violation is found in the model, that is not a valid implementation behaviour. This is due to over-approximation.

There are two main types of properties that can be expressed using LTL:

- *safety properties* usually state that something bad never happens ($\mathbf{G}\ \neg\phi$), while
- *liveness properties* state that something good eventually happens ($\mathbf{F}\ \phi$).

Some of the properties we can verify for the FTP server:

- The server will infinitely often be in its accept state. One way to check this is to use mark the specific state with a progress label. A progress label states the requirement that the labeled global state must be visited infinitely often in any infinite system execution. Any violation of this requirement can be reported by the verifier as a non-progress cycle.
- If a client request a file retrieval, the file will eventually arrive at the client. This is a response property: $\mathbf{G}\ (p \rightarrow (\mathbf{F}\ q))$, where p is the atomic proposition indicating that a request has been made, and q the atomic proposition indicating that the file has been received.

To verify the server we introduce non-determinism into the system environment so that all possible system behaviours are checked. For example, `client` processes are non-deterministically instantiated, representing any possible system load. Also, non-determinism is introduced into the `clientPI` process itself so that it drives the server through all possible behaviours, and even some that are not valid.

4.2.3 Implementation

Using the GENERATOR a complete implementation is generated for the server. Using a standard FTP client we can connect to the server and perform any of the standard file related functions. For the implementation the generated server is made to "listen" on a different port as the standard port 21, since this is usually already assigned to an installed FTP server.

Performance Results

To examine the efficiency of the generated implementation, it was compared to a mature Linux FTP server, GPro ftpd [90]. Table 1 contains the results of a performance comparison of file transfer using the generated server (Gen ftpd), and the Linux server (GPro ftpd). The experiment was set up over a private network with no other traffic, and each measurement represents the average result of retrieving the specific file ten times from the corresponding server.

This experiment clearly shows that the generated code is just as efficient as that of a mature and manually optimised implementation. There is thus no reason not to use this approach to

	318K	112M	500M	1G	(bytes)
GPro ftpd	0.153	12.1	58.6	130	(s)
	2330	9440	8730	8180	(KB/s)
Gen ftpd	0.139	12.7	59.1	123	(s)
	2570	9040	8670	8490	(KB/s)

Table 1: File transfer results.

develop reactive systems, since it preserves the benefits of model checking while addressing its limitations.

Chapter 5

Conclusions

The general goal of this thesis was to investigate a technique that would allow model checking to be directly integrated into the software development process, preserving the benefits of model checking while addressing some of its limitations. This in effect addresses both issues central to software engineering, cost and reliability. Specifically, we wanted to address the *model construction problem.* A technique was developed that allows a complete executable implementation to be generated from an enhanced model specification. This included the development of a program, the GENERATOR, that completely automates the generation process. As this approach effectively combines the modelling and implementation steps, the verification effort is perfectly in step with the development process, which means *timing* and *relevance* are no longer issues. This addresses one half of the model construction problem.

In addition, it was also illustrated how structuring the specification as a transitions system formally separates the control flow from the details of manipulating data. This simplifies the verification process which is focused on checking control flow in detail. By combining this structuring approach with automated implementation generation we ensure that the verified system behaviour is preserved in the actual implementation. This addresses the other half of the model construction problem. An additional benefit is that data manipulation, which is generally not suited to model checking, is restricted to separate, independent code fragments that can be verified using verification techniques for sequential programs. These data manipulation code segments can also be optimised for the implementation without affecting the verification of the control structure; in most cases it is only necessary to take into account all possible outcomes of an operation, and not the actual detail of the operation itself. This approach of course does not lend itself equally well to all kinds of applications, with event driven systems being a natural fit. Future work could include the investigation of the application of this approach to different types of applications.

As part of this investigation an FTP server was implemented, illustrating that the technique can be used to successfully implement an efficient reactive system of realistic size. Although it was outside the scope of this thesis to use this technique to implement an industrial size application and follow it through its complete life cycle, this technique shows enough similarities with the approach taken by FeaVer (Subsection 2.5.4) to suggest that it will scale equally well. Whereas FeaVer extracts a model from implementation source code and uses a lookup table to map source statements to their abstract model representation, the Generator generates an implementation source from an extended model and uses embedded code fragments to map model operations to their implementation counterparts.

Even though the approach presented in this thesis has shown promising results, it is by no stretch of the imagination the proverbial "silver bullet". It is important to recognise that even though the majority of the control structure is formally and automatically translated, the mapping between implementation and model specific code segments are not proven formally equivalent by the Generator. In some cases it is possible to use techniques such as weakest preconditions to achieve this, while in other cases it might be possible to use model checking or theorem proving to automatically prove that a mapping is correct. Despite this limitation, this approach may still offer higher reliability than alternatives, because the gap between the model and the implementation has been reduced to gaps between smaller code fragments.

Furhermore, this approach still requires that the user defines the complete environment of the system, as well as all of the mappings. This means that the possibility of human error is still not completely removed, and there is still extra work involved as opposed to just implementing the application. Our goal, however, was not to develop a completely automated verification system, but simply to improve the reliability of the process through the use of the model–implementation mappings we have described. We conclude that for the development of concurrent and reactive systems, this approach will outperform traditional testing methods, satisfying our goals.

Although this first experiment is seen as successful, there are a number of possible future enhancements that would make the Generator more user-friendly and effective as a development tool:

- As a research prototype, one of the most important future activities would be to verify the Generator more aggressively, and to apply it to as many examples as possible to increase confidence in its reliability.

- It would be useful if the Generator could also automate the verification of the data manipulation code segments. One option would be to automatically generate test cases.
- The Generator currently does not interpret the actual C code, and although this is not necessary for correct operation (the C compiler will report any errors), having the Generator parse the C code will be more user-friendly, since parse errors will be detected earlier on. This will also enable the Generator to interpret the C code in terms of the complete specification.
- The Generator could be extended to perform automatic dependency checks and warn the user if implementation specific code contains an assignment to a variable without a model counterpart, for example. In other words, the Generator may be extended to warn the user if the abstract-concrete mappings are incomplete.
- The current version of the Generator has a general approach for handling the mapping between implementation and model specific representations of objects and the functions that operate on them. This was done so as not to limit the tool by binding it to a specific application. It would however be possible to add more extensions to Promela that would indicate to the Generator that a certain channel and the operations on it should automatically be translated to a specific implementation representation, or define standard mappings. For large applications with long lifetimes, it would conceptually be worth the initial effort to build such translation libraries.
- A much more ambitious suggestion is the development of an integrated, graphical development environment, able to simulate and debug the complete specification, including the C code segments. It would conceptually be possible to step the execution of the model and implementation specific code segments in tandem, using an approach similar to a visual debugger. This would allow counterexamples produced for the model to be mapped to the implementation. This way it would be easy to identify false positives. One way to achieve all of this would be to integrate the Generator into Spin and enhance Spin to support these features. This will only be a viable option if it could be done without branching off from the main development tree of Spin, otherwise it could eventually become outdated and irrelevant. The Generator was specifically implemented as a standalone tool to avoid this.

As this thesis has demonstrated, it is possible to bridge the specification gap without sacrificing efficiency. In fact, there are good arguments to support the idea that the Generator and other tools like it can improve the productivity of software developers. This is true not only for mission-critical projects, for which correctness is vital, but also for other software. Observing that the focus of CPU manufacturers has shifted from increasing clock frequency to increasing the computing cores in order to evolve computing power and efficiency, it is safe to say that concurrent software will become more prevalent in almost all system types to fully take advantage of the hardware. Model checking has been proven to be one of the most effective verification methods when it comes to concurrent systems, and the next step is to evolve the technology into an effective and user-friendly developer tool. Whether the technology is based on model extraction, program generation, or directly model checking software, it is likely that such tools will in the future become part of the standard software engineering toolkit.

Bibliography

[1] A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall International (UK) Ltd., 1994. Translated by J. Plaice.

[2] T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the 7th International Spin Workshop on Model Checking and Software Verification*, pages 113–130. Springer-Verlag, August, September 2000.

[3] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the 8th International Spin Workshop on Model Checking Software*, pages 103–122, May 2001.

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

[5] A. Basu. *A Language-Based Approach to Protocol Construction*. PhD thesis, 1998.

[6] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A Language-Based Approach to Protocol Construction. January 1997.

[7] A. Basu, T. von Eicken, and G. Morrisett. Promela++: A Language for Correct and Efficient Protocol Construction. In *Proceedings of the IEEE INFOCOM'98 Conference on Computer Communications*, pages 455–462, March-April 1998.

[8] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207, 1999.

[10] J. P. Bowen and M. G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(3):34–41, 1995.

[11] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington, and W. Visser. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in System Design*, 25, September 2005.

[12] G. Brat, K. Havelund, S. Park, and W. Visser. Java Pathfinder: Second Generation of a Java Model Checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000. post-CAV 2000.

[13] D. L. Bruening. Systematic Testing of Multithreaded Java Programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[14] J. J. D. Bull. A Comparison of Two Different Model Checking Techniques. Master's thesis, University of Stellenbosch, December 2003.

[15] J. J. D. Bull and P. J. A de Villiers. Using SPIN to Verify Protocols at the Implementation Level. In *Proceedinngs of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 195–204, September 2002.

[16] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, 1997.

[17] E. Clarke, D. Kroenig, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 168–176. Springer-Verlag, March, April 2004.

[18] E. M. Clarke. ANSI-C Bounded Model Checker - User Manual, August 2003.

[19] E M. Clarke and S. Berezin. Model Checking: Historical Perspective and Example (Extended Abstract). In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 18–24. Springer-Verlag, 1998.

[20] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Workshop on Logic in Programs*, volume 131, pages 52–71. Springer-Verlag, May 1981.

[21] E. M. Clarke, E. A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[22] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV'93)*, volume 697, pages 450–462. Springer-Verlag, 1993.

[23] E. M. Clarke, O. Grumberg, and D. Long. Model Checking. In *NATO ASI International Summer School on Deductive Program Design*, volume 152 of *F*. Springer-Verlag, 1994.

[24] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing Checkable Properties of Dynamic Systems: The Bandera Specification Language, 2000. Submitted for publication. A shorter version of this paper appeared in the 2000 Spin Workshop.

[25] J. C. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zeng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, June 2000.

[26] C. Cornes, J. Courant, J. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, and C. Parent et al. The Coq Proof Assistant - Reference Manual V 5.10.

[27] D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. 19(2):253–291, 1997.

[28] D. Dams, O. Grumberg, and R. Gerth. Generation of Reduced Models for Checking Fragments of CTL. In *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV'93)*, pages 479–490, 1993.

[29] C. DeMartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 5th and 6th International Spin Workshops*, pages 261–276, July, September 1999.

[30] C. DeMartini, R. Iosif, and R. Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software - Practice and Experiece*, 29(7):577–603, 1999.

[31] E. W. Dijkstra. Notes on Structured Programming. Circulated privately: `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`, April 1970.

[32] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.

[33] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported Program Abstraction for Finite-state Verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, May 2001.

[34] E. A. Emerson and A. P. Sistla. Symmetry and Model Checking. In *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV'93)*, pages 463–478, 1993.

[35] K. Etessami and G. J. Holzmann. Optimizing Büchi Automata. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*, pages 153–167. Springer-Verlag, 2000.

[36] J. Geldenhuys and P. J. A. de Villiers. Runtime Efficient State Compaction in Spin. In *Proceedings of the 5th and 6th International Spin Workshops on Theoretical and Practical Aspects of Spin Model Checking*, pages 12–21, 1999.

[37] J. Geldenhuys and A. Valmari. A Nearly Memory-optimal Data Structure for Sets and Mappings. In *Proceedings of the 10th International Spin Workshop on Model Checking of Software*, pages 136–150. Springer-Verlag, May 2003.

[38] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15th IFIP Symposium on Protocol Specification, Testing, and Verification (PSTV'95)*, pages 3–18. Chapman & Hall, 1995.

[39] P. R. Glück and G. J. Holzmann. Using Spin Model Checking for Flight Software Verification. In *Proceedings of the 2002 Aerospace Conference*, volume 1, pages 105–113, March 2002.

[40] P. Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97)*, pages 172–186, June 1997.

[41] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *Formal Methods in System Design*, 7(3):227–241, November 1995.

[42] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 305–326, 1991.

[43] Patrice Godefroid. Model checking for Programming Languages using VeriSoft. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

[44] Patrice Godefroid. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design*, 26(2):77–101, March 2005. journal version of POPL'97 paper.

[45] M. J. C. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report 68, Computer Laboratory, University of Cambridge, July 1985.

[46] J. C. Grégoire. State Space Compression with GETSs. pages 90–108, August 1997.

[47] A. Groce and W. Visser. Heuristics for Model Checking Java Programs. *International Journal on Software Tools for Technology Transfer*, 4(6), December 2004.

[48] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[49] D. Harel and A. Pnueli. On the Development of Reactive Systems. *Logics and Models of Concurrent Systems*, F(13):477–498, 1985.

[50] J. Hatcliff and M. B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'01)*, pages 39–58. Springer-Verlag, August 2001.

[51] K. Havelund. Java pathfinder, a translator from java to promela. In *Proceedings of the 5th and 6th International Spin Workshops on Theoretical and Practical Aspects of Spin Model Checking*, page 152. Springer-Verlag, 1999.

[52] K. Havelund, M. R. Lowry, and J. Penix. Formal Analysis of a Space-Craft Controller Using Spin. In *Proceedings of the 4th International Spin Workshop*, November 1998.

[53] K. Havelund and T. Pressburger. Model Checking Java Programs using Java Pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris France, 1998.

[54] K. Havelund and W. Visser. Program Model Checking as a New Trend. *International Journal on Software Tools for Technology Transfer*, 4(1), October 2002.

[55] T. Henzinger, R. Jhala, and R. Majumdar. Lazy Abstractions. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, January 2002.

[56] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast, May 2003.

[57] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[58] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[59] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[60] G. J. Holzmann. On Limits and Possibilities of Automated Protocol Analysis. In *Proceedings of the 7th International Conference on Protocol Specification, Testing, and Verification (PSTV'87)*, pages 339–344. North-Holland, 1987.

[61] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[62] G. J. Holzmann. State compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings of the 3rd Spin Workshop*, April 1997.

[63] G. J. Holzmann. Logic Verification of ANSI-C Code with SPIN. In K. Havelund, editor, *Proceedings of the 7th International Spin Workshop on Model Checking and Software Verification*, pages 131–147. Springer-Verlag, September 2000.

[64] G. J. Holzmann. From Code to Models. In *Proceedings of the 2nd International Conference on Applications of Concurrency to System Design*, pages 3–10, June 2001.

[65] G. J. Holzmann. *The* SPIN *Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.

[66] G J Holzmann. Software Model Checking with SPIN. In *Advances in Computers*, volume 65, pages 77–108. Elsevier, July 2005.

[67] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. In *Proceedings of the 12th International Conference on Protocol Specification, Testing, and Verification (PSTV'92)*, pages 349–363. North-Holland, 1992.

[68] G. J. Holzmann and R. Joshi. Model-Driven Software Verification. In *Proceedings of the 11th International Spin Workshop on Model Checking of Software*, pages 77–92. Springer-Verlag, April 2004.

[69] G. J. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proceedings of the 7th International Conference on Formal Description Techniques*, pages 197–211. Chapman & Hall, 1994.

[70] G. J. Holzmann and D. Peled. Partial Order Reduction of the State Space. 1995.

[71] G. J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.

[72] G. J. Holzmann and M. H. Smith. A Practical Method for the Verification of Event Driven Systems. In *Proceedings of the 21st International Conference on Software Engineering*, pages 597–608, May 1999.

[73] G. J. Holzmann and M. H. Smith. Software Model Checking: Extracting Verification Models from Source Code. *Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 481–497, October 1999.

[74] G. J. Holzmann and M. H. Smith. Automating Software Feature Verification. *Bell Labs Technical Journal*, 5(2):72–87, April, June 2000. Special issue on software complexity.

[75] F. Huch. Verification of Erlang Programs using Abstract Interpretation and Model Checking. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, pages 261–272. ACM Press, 1999.

[76] F. Huch. Model Checking Erlang Programs — Abstracting the Context-Free Structure, 2001. Manuscript, Private communication.

[77] C. N. Ip and D. L. Dill. Better Verification Through Symmetry. In *Computer Hardware Description Languages and their Applications*, pages 87–100. Elsevier Science Publishers, 1993.

[78] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model Checking for Managers. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 92–107. Springer-Verlag, 1999.

[79] P. Kars. The Application of PROMELA and SPIN in the BOS Project. In *Proceedings of the 2nd Spin Workshop*, August 1996.

[80] J. Katoen. *Concepts, Algorithms and Tools for Model Checking*. Arbeitsberichte der Informatik, Friedrich-Alexander-Universitaet Erlangen-Nuernberg. Gruner Druck GmbH, 1999.

[81] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, December 1997.

[82] S. Löffler. From Specification to Implementation: A Promela to C Compiler. Semesterarbeit, Unversität Stuttgart, 1996.

[83] S. Löffler and A. Serhrouchni. Creating a Validated Implementation of the Steam Boiler Control. In *Proceedings of the 3rd Spin Workshop*, April 1997.

[84] S. Löffler and A. Serhrouchni. Creating Implemenations from Promela Models. In *Proceedings of the 2nd Spin Workshop*, pages 72–80, 1997. Held August 1996, DIMACS Series No. 32.

[85] K. L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.

[86] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[87] R. Milner. A Calculus of Communication Systems. *Lecture Notes in Computer Science*, (92), 1980.

[88] R. Milner. *Communication and Concurrency*. 1989.

[89] G. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965.

[90] J. Morrissey and T.J. Saunders. ProFTPD. http://www.proftpd.org/.

[91] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. Cmc: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, December 2002.

[92] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96)*, pages 411–414. Springer-Verlag, July-August 1996.

[93] D. Y. W. Park, U. Stern, J. U. Sakkebaek, and D. L. Dill. Java Model Checking. In *Proceedings of the 1st International Workshop on Automated Program Analysis, Testing and Verification*, pages 253–256, June 2000.

[94] B. Parreaux. Difference Compression in Spin. In *Proceedings of the 4th International Spin Workshop*, November 1998.

[95] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV'94)*, pages 377–390. Springer-Verlag, 1994.

[96] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 488–497, June 2000.

[97] J. Penix, W. Visser, C. Pasareanu, E. Engstrom, A. Larson, and N. Weininger. Verifying Time Partitioning in the DEOS Scheduling Kernel. *Formal Methods in System Design*, 26(2), March 2005.

[98] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[99] J. Postel and J. K. Reynolds. RFC 854: Telnet Protocol Specification, May 1983.

[100] J. Postel and J. K. Reynolds. RFC 959: File Transfer Protocol (FTP), October 1985.

[101] J-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–351. Springer-Verlag, 1982.

[102] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann. Validating Requirements for Fault Tolerant Systems using Model Checking. In *Proceedings of the 3rd IEEE International Conference on Requirements Engineering*, pages 4–14, April 1998.

[103] U. Stern and D. L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Proceedings of the IFIP WH 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'95)*, pages 206–224. Springer-Verlag, 1995.

[104] S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *Proceedings of the 7th International Spin Workshop on Model Checking and Software Verification*, pages 224–244. Springer-Verlag, August, September 2000.

[105] S. K. Rajamani T. Ball. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 1–3, January 2002.

[106] A. Valmari. Stubborn Sets for Reduced State Space Generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 1–22, 1989.

[107] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the Symposium on Logic in Computer Science (LICS'86)*, pages 322–331, 1986.

[108] M. Y. Vardi and Pierre Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.

[109] W. Visser and H. Barringer. Memory Efficient State Storage in SPIN. In *Proceedings of the 2nd Spin Workshop*, August 1996.

[110] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 3–11, September 2000.

[111] P. Wolper. Verification: Dreams and Reality, Febuary 1998. Inaugural Lecture: http://www.montefiore.ulg.ac.be/ pw/cours/psfiles/francqui-lect1.ps.

[112] P. Wolper and D. Leroy. Reliable Hashing Without Collision Detection. In *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV'93)*, pages 59–70, 1993.

[113] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about Infinite Computation Paths. In *Proceedings of the 24th IEEE Symposium on the Foundations of Computer Science*, pages 185–194, 1983.