Efficiency Issues in the Design of a Model Checker

A THESIS PRESENTED IN PARTIAL FULFILMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

AT THE UNIVERSITY OF STELLENBOSCH

By

Jaco Geldenhuys

November 1999

Supervised by: P. J. A. de Villiers

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: ..... Date: .....

Summary

A *model checker* is a program that verifies, without human assistance, that the formal description of a system has specified, desirable properties. The development of model checking algorithms is an active area of research, but most implementations are still prototypical in nature. In consequence, knowledge about the design and implementation of a practical, efficient model checker is limited.

In this thesis the most important design decisions involved in creating an efficient on-the-fly model checker are identified and discussed. In short, there are three major tasks:

1. the generation of program states,
2. the detection of revisited states, and
3. the representation of states.

In all three cases the central goal is to generate as many states as possible and to generate states as fast as possible. For each task, alternatives are described and compared.

The discussion of design issues is further supported in two ways. First, a detailed design and implementation for a model checker is described to illustrate how design decisions affect each other and ultimately the implementation. Second, the design arguments, based on more or less realistic models, are validated through a thorough study of the performance of the various components of the model checker.

Afrikaans summary

'n *Modeltoetser* is 'n program wat vasstel of die formele beskrywing van 'n stelsel oor wenslike, vooraf-gespesifiseerde eienskappe beskik. Die ontwikkeling van algoritmes vir hierdie doel word aktief nagevors, maar in die meeste gevalle is implementasies van modeltoetsers van 'n bloot prototipiese aard. Gevolglik is kennis oor die ontwerp en implementering van 'n praktiese, effektiewe modeltoetser so skaars soos hoendertande.

Hierdie tesis bespreek die belangrikste ontwerpsbesluite in die ontwikkeling van 'n effektiewe modeltoetser. Drie hooftake word geïdentifiseer:

1. die voortbrengs van state (programtoestande),
2. die herkenning van reeds bekende state, en
3. die interne voorstelling van state.

In al drie gevalle is die belangrikste doelwit om so veel as moontlik state voort te bring, en om state so vinnig as moontlik voort te bring. Vir elke taak word alternatiewe bespreek en vergelyk.

Die bespreking word verder op twee maniere ondersteun. Eerstens word 'n modeltoetser se ontwerp en implementasie in detail beskryf om die invloed van ontwerpsbesluite op mekaar en op die uiteindelike implementasie te illustreer. Tweedens word die argumente, tot dusver gebaseer op redelike aannames, gevalideer deur 'n deeglike studie van die werkverrigting van die modeltoetser se onderskeie onderdele.

Acknowledgements

I have accrued quite a debt of gratitude while writing this thesis:

- First and foremost, I would like to thank my supervisor Pieter de Villiers for advice, guidance and general kindness *far* beyond the call of duty.
- I am indebted to all the members, past and present, of the Hybrid/Gneiss project for providing a stimulating work environment.
- My thanks to the Department of Computer Science at Stellenbosch University and the Software Systems Laboratory at Tampere University of Technology who have been generous with their time to allow me to finish this work.
- I gratefully acknowledge the financial support I received from the Foundation for Research and Development, the Harry Crossley Trust and the Stellenbosch 2000 Trust.

Finally, thank you to my forbearing family and friends for their loyal support and continuous encouragement. Without you, ...

For MMSG and MEJO

Contents

1 Introduction **1**

2 An overview of model checking **4**

2.1 Finite transition systems 4

2.2 Temporal logic 7

2.2.1 CTL 8

2.2.2 Correctness specifications 9

2.3 Model checking algorithms 9

2.3.1 Automata-based algorithms 10

2.3.2 Structure-based algorithms 11

2.3.3 Avoiding redundant work 11

2.4 The state explosion problem 12

2.5 Fairness 13

3 Design issues **16**

3.1 Requirements 18

3.2 State generation 20

3.2.1 The execution of transitions . . . 21
3.2.2 Non-determinism . . . 26
3.2.3 Communication . . . 28
3.2.4 Dynamic process creation . . . 32
3.3 The detection of revisited states . . . 33
3.3.1 State caching . . . 33
3.3.2 Bitstate hashing . . . 35
3.3.3 State caching v. bitstate hashing . . . 35
3.3.4 Implicit representations . . . 36
3.4 The representation of states . . . 36
3.4.1 General issues . . . 37
3.4.2 Representation techniques . . . 39
3.4.3 State compaction . . . 41
3.5 Fairness . . . 44
3.5.1 Impartiality . . . 44
3.5.2 Strong fairness . . . 45
3.5.3 Weak fairness . . . 46

4 Implementation of a model checker **47**

4.1 State generation . . . 48
4.1.1 Procedures `Execute` and `Backtrack` . . . 49
4.1.2 The architecture of the abstract machine . . . 51
4.1.3 The structure of the stack . . . 62

4.2 State caching 63
4.3 State compaction 65
4.4 The implementation of fairness 66
4.4.1 Simplifying assumptions 67
4.4.2 Procedures `Push` and `Pop` 67
4.4.3 Tarjan's original algorithm 68
4.4.4 Integration into `Push` and `Pop` 71

5 Evaluation **76**

5.1 State compaction 78
5.1.1 Number of compaction operations 78
5.1.2 Validating without compaction 79
5.2 State caching 80
5.2.1 The influence of cache size 81
5.2.2 The influence of tolerance parameters 84
5.3 The cost of fairness 86
5.3.1 The detection of SCCs 87
5.3.2 Model checking overhead 88
5.4 Interpretation 89
5.4.1 Interpreter v. pre-compiled state generator 90
5.4.2 The composition of the instruction set 92
5.5 Overall performance 94
5.6 Summary 96

6 Conclusion **97**

A Model source code **99**

A.1 Model of seven dining philosophers (DP7) 100

A.2 Model of the elevator with three floors (EL3) 102

A.3 Model of the process scheduler with one process (PS1) 105

A.4 Model of the sliding window protocol with window size one (SW1) 108

B Model analysis details **111**

B.1 Measurements for dining philosophers models (DPn) 113

B.2 Measurements for elevator models (ELn) 114

B.3 Measurements for process scheduler models (PSn) 115

B.4 Measurements for sliding window protocol models (SWn) 116

C Abstract instruction set **117**

C.1 Arithmetic instructions 117

C.2 Memory manipulation instructions 118

C.3 List manipulation instructions 118

C.4 Communication instructions 118

C.5 Control flow instructions 119

C.6 Miscellaneous instructions 119

D The ESML modelling language **120**

D.1 Constant, type and variable definitions 121

D.2 Expressions 123

D.3 Commands 124

D.4 Processes and models 127

D.5 A grammar for ESML 128

Bibliography **132**

Bibliographic crossreference **138**

List of Tables

5.1 Models selected for performance measurements 77

5.2 Count of compaction operations for the sliding window protocol models (SWn) 78

5.3 Performance of the model checker with and without compaction 80

5.4 Overhead of SCC detection 87

5.5 SCC sizes and stack requirements 88

5.6 Four verification runs of the process scheduler model (PS2) 89

5.7 Execution profile of the model checker for EL4 91

5.8 Instruction frequency for DP9, EL4 and PS3 92

5.9 Comparison of Promela and ESML models 95

List of Figures

2.1 State graph of FTS M 6

2.2 State graph representing the mutual exclusion problem 6

2.3 Automaton that accepts $AG(t \Rightarrow AF(c))$ 10

2.4 Reduced state graph of the mutual exclusion problem 13

2.5 State graph with fairness problems 13

3.1 Exhaustive depth-first exploration of a simple state graph 19

3.2 The transition table and code generated for process `Counter` 23

3.3 Abstract code for process `Counter` 25

3.4 ESML model of the mutual exclusion problem 27

3.5 Interaction of channel queues and `POLL` commands 30

3.6 Process state enumeration 41

4.1 Module structure of the model checker 48

4.2 Procedure `Execute` 49

4.3 Procedure `Backtrack` 51

4.4 Key data structures of the interpreter 52

4.5 Procedure `Step` 53

4.6 Implementation of the `add` instruction 55

4.7 Code generated for the assignment `x := x - 1` 56

4.8 Implementation of the `popVariable` instruction 57

4.9 Code generated for an `IF` command 58

4.10 Code generated for an `POLL` command 60

4.11 Implementation of the `index` instruction 61

4.12 Illustration of the operation of the delta store 63

4.13 Definition of the `StackEntry` type 63

4.14 Definition of the `CacheEntry` type 64

4.15 Modified definition of the `StackEntry` type 64

4.16 Procedure `GetValue` 66

4.17 Procedure `UpdateValue` 66

4.18 Procedure `Push` 68

4.19 Procedure `Pop` 68

4.20 Depth-first search of state graphs 69

4.21 Tarjan's original algorithm 70

4.22 SCC and depth-first stack 72

4.23 Modified procedure `Push` 74

4.24 Modified procedure `Pop` 74

5.1 Influence of cache size on performance for the elevator model (EL3) 82

5.2 Number of visits per state for the elevator model (EL3) 83

5.3 Details of influence of cache size on transitions for the elevator model (EL3) . . 84

5.4 Insertion algorithm for the state cache 85

Chapter 1

Introduction

Computer software is playing an ever-increasing role in our lives. At the same time, software is growing in complexity to meet the demands of greater scale and functionality. And yet we are still trapped in the midst of the software crisis that was identified in the mid sixties. Despite advances in the development of software, software remains expensive to produce and its quality remains difficult to measure and ensure. While we can sometimes tolerate a degree of unreliability, there are many cases where subtle errors in software can cause a catastrophic loss in money, time and even human life.

One recourse for the development of systems for which correctness is critical, is the use of formal methods. Over the last twenty years formal methods—the systematic application of mathematical rigour to program development—has met with growing success. One aspect of this has been the evolution of computer-aided verification, and one of the most successful of these techniques is model checking.

A model checker is a program that verifies, without human assistance, that the formal description of a system has specified, desirable properties. It operates by investigating all the possible states that the system can assume. The success of model checking is due to many factors: once the user has specified the system and its correctness properties the process is fully automatic and requires no expert or theoretical knowledge; it is fast compared to other methods of proving correctness; in many cases a model checker can provide, at little additional cost, witnesses to show why a property holds and even more useful counterexamples to show why a property fails

to hold; model checkers allow users to specify systems in an intuitive way and logics can easily express many interesting concurrency properties.

Safety-critical software are often instances of *reactive systems*: programs that do not terminate but engage in continuous interaction with their environment. Correctness is important because reactive systems are often widely used (for example, communication protocols), perform life-critical functions (aircraft control systems), or are expensive to produce and modify (embedded control software for microprocessors). Reactive systems are complicated by the fact that they usually consist of several processes executing concurrently. Model checkers are suitable tools for the development of correct reactive systems.

The study of model checking algorithms is an active area of research, but when it comes to implementations the focus of most efforts falls on experimental purposes with more attention paid to quickly obtaining a working prototype, and less to efficiency concerns. In consequence, documentation about the design and implementation of a practical, industrial-strength model checker is limited. (A singular exception is the SPIN system, which is arguably the most widely used and best documented model checker at present [34].) As research in this field advances, the application of model checking to meaningful real-world problems to obtain useful results is becoming more viable, and consequently knowledge about the design of model checkers is becoming increasingly valuable.

The goal of this thesis

There are several approaches to the model checking problem; we will concentrate on on-the-fly model checking, a suitable technique for the verification of software designs. In this thesis the most important design decisions involved in creating an efficient on-the-fly model checker are identified and discussed. In short, there are three major tasks:

1. the generation of program states,
2. the detection of revisited states, and
3. the representation of states.

In all three cases the central goal is to generate (and store) as many states as possible and to generate states as fast as possible. For each task, alternative techniques are described and compared.

The discussion of design issues is further supported in two ways. First, a detailed design and implementation of a model checker made by the author is described to demonstrate how design decisions affect each other and ultimately also the implementation. Second, a thorough evaluation of the model checker's performance lends weight to the central arguments by illustrating the actual cost of design choices.

Thesis outline

Chapter 2: An overview of model checking introduces finite transition systems and temporal logic, describes the model checking problem and presents a brief overview of on-the-fly model checking. The major obstacle in model checking software is that the number of states grow exponentially in the number of variables and processes; this so-called *state explosion* problem is discussed, and lastly the issue of fairness is addressed.

Chapter 3: Design issues forms the core of the thesis. It defines the major components of a model checker and their interface with the model checking algorithm. For each component its critical issues are described and different design alternatives are presented and evaluated.

In ***Chapter 4: Implementation of a model checker*** the design of an actual model checker is described down to the level of implementation, where appropriate. The model checker uses an on-the-fly algorithm for a subset of CTL, a cache of compacted states, and includes support for strong fairness. A central feature of this system is that states are generated by an abstract machine interpreter.

Chapter 5: Evaluation presents the results of experiments conducted to measure the performance of various components of the model checker. This includes the performance effects of state compaction, state caching, SCC detection (for strong fairness), and interpretation. As a benchmark, the model checker is compared to the SPIN system.

Lastly, a summary and conclusions are given in ***Chapter 6: Conclusion***.

Chapter 2

An overview of model checking

A model checker is a program that verifies automatically that the formal description of a system has specified, desirable correctness properties. This process involves three key ingredients:

- a formalism for describing the behaviour of systems in a formal and precise way,
- a formalism for expressing the properties to be verified, and
- an algorithm that will perform the verification.

These requirements are addressed in the first three sections of this chapter.

Unfortunately, model checkers cannot verify the correctness properties of arbitrarily large systems. Compared to actual implementations, the systems that can be verified are relatively tiny. This limitation, known as the *state explosion problem*, is discussed in Section 2.4. A secondary aspect of verification important for checking certain properties is *fairness*; this is the topic of Section 2.5.

2.1 Finite transition systems

To study the properties of a concurrent system, its behaviour must be described in a precise way: formal notation is needed. A suitable notation is a finite transition system (FTS). An

FTS is a tuple $M = (S, T, R, \hat{s})$ where

- S is a finite set of states,
- T is a finite set of transitions,
- $R \subseteq S \times T \times S$ is the transition relation, and
- $\hat{s} \in S$ is the initial state.

A *state* is a canonical description of a system at a specific moment in time. It uniquely identifies the values of location counters, variables, queue contents and other data structures. A *transition* is an atomic step that makes a system change from one state into another. When $(s, t, s') \in R$ it means that transition t is *enabled* in state s and its execution will change the state of the system from s to s'. This is abbreviated as $s \xrightarrow{t} s'$ and state s' is called a *successor state* of s. The set of all enabled transitions in state s is denoted by $en(s)$. A *path*, or sometimes *execution path*, is a (possibly infinite) sequence of states $\sigma = s_0, s_1, s_2, \ldots$ so that for all $i > 1$ there is a transition t_i such that $(s_{i-1}, t_i, s_i) \in R$. In the case of a finite path, the sequence has a last state s_n with $n > 0$, so that $(s_{i-1}, t_i, s_i) \in R$ only for $1 < i \leq n$, and the length of the path is said to be n. State s' is *reachable* from state s if there is a finite path $\sigma = s_0, s_1, \ldots, s_n$ so that $s = s_0$ and $s' = s_n$. This is written as $s \xrightarrow{\sigma} s'$.

An FTS is used to describe the behaviour of a concurrent system that results from the interaction of one or more *processes*. The set of transitions can be partitioned into a set $\tau = \{T_0, T_1, \ldots, T_m\}$. Each T_k is the subset of T that contains the transitions that belong to process k.

An FTS can be interpreted as a directed graph by taking S as the set of vertices and R as the set of labeled edges. This is called a *state graph*, and it is useful for visualising the behaviour of the system it describes. When an FTS is represented in this way, the initial state is indicated by a source-less arrow pointing at its vertex.

Figure 2.1 shows an example of a state graph. It describes a process that starts in state n (for noncritical). It then moves to state t (for trying). In this state the process tries to enter state c (for critical), where it performs critical operations before returning to state n.

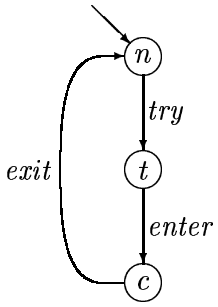


Figure 2.1: State graph of FTS M

The transitions that are executed are *try*, *enter*, and *exit*, and the underlying FTS is $M = (\{n, t, c\}, \{try, enter, exit\}, R, n)$. The set $R = \{(n, try, t), (t, enter, c), (c, exit, n)\}$.

FTS M describes the behaviour of one process; the concurrent behaviour of two such processes is depicted in Figure 2.2. The transition labels have been omitted for the sake of clarity. The state names indicate the state of each of the two processes. For example, in state tc the first process is in its trying state, while the second process is in its critical state. This state graph represents an instance of the mutual exclusion problem: only one process at a time is allowed to enter its critical state. For this reason, the state graph does not contain all possible combinations of states of M: state cc is missing.

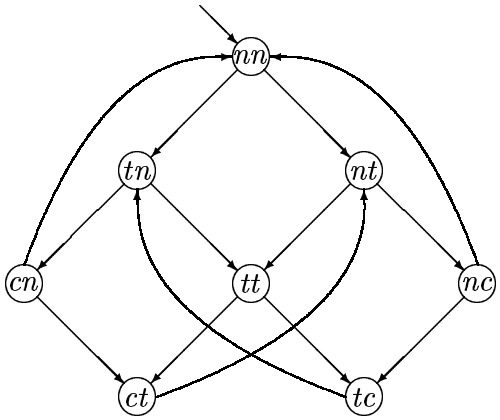


Figure 2.2: State graph representing the mutual exclusion problem

2.2 Temporal logic

Classical propositional logic has proven successful for reasoning about programs that accept input, perform a computation to transform it, and to finally yield output. It is convenient to view these programs as relations from an initial state to a final state. One example of this approach is Dijkstra's weakest precondition calculus [16]. However, concurrent systems cannot be adequately described in this way. For these systems the use of temporal logic is recommended, since it can express the ordering of events in time without introducing time explicitly. The use of temporal logic for reasoning about concurrent systems was pioneered by Pnueli in 1977 [45].

There are two varieties of temporal logic: linear time and branching time. *Linear time* temporal logic (for example, (Propositional) Linear Time Logic (LTL) [41]) is concerned with logical properties of a single execution path of a system. Linear temporal logic formulas express properties that must hold for every possible path starting at the initial state. Since each state may have several possible successor states there may be many different paths that start in the initial state. Any particular path is one "branch" of the tree of all possible future states. *Branching time* temporal logic (for example, Computation Tree Logic (CTL) [10]) can distinguish between properties that must hold in all possible futures and those that must hold in at least one possible future.

Since their very formulation, there has been some debate over the relative merits of linear and branching time temporal logic [18, 28, 39, 42]. In an attempt to resolve this question, Emerson formulated a new temporal logic CTL* that contains both LTL and CTL [18, 19]. He showed that the two logics are not comparable in expressive power. In other words, LTL can express properties that CTL cannot, and vice versa.

Although both LTL and CTL are widely used for model checking, in the rest of this thesis correctness specifications are expressed in CTL.

2.2.1 CTL

CTL was introduced by Clarke and Emerson [10] in 1981. For our purposes, an informal description suffices—for a thorough logical treatment the reader is referred to [17]. Formally, the semantics of CTL is defined with respect to a *Kripke structure*, but the above definition of an FTS is almost sufficient: the only addition is that of a set of atomic propositions. Each element of this set is a proposition that has a value of either true or false in every state. An example of an atomic proposition is $x \geq 5$: this formula is true in some states (those where x has a value greater than or equal to five) and false in all others. Often atomic propositions are not specified explicitly, but are referred to by names such as "p" and "q".

CTL contains all of the classical propositional calculus: atomic propositions (p, q, r, ...), binary operators ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$), and negation ($\neg$). In addition, the logic contains eight temporal operators. Each of these consists of two symbols: one path quantifier, either A ("for all execution paths") or E ("for at least one execution path"), and one state quantifier, G ("always"), F ("eventually"), X ("next state"), or U ("until"). The resulting operators are interpreted in the following way:

- $AG(\phi)$ along all paths, ϕ holds in all states
- $EG(\phi)$ along some path, ϕ holds in all states
- $AF(\phi)$ along all paths, there is a state where ϕ holds; ϕ is *inevitable*
- $EF(\phi)$ along some path, there is a state where ϕ holds; ϕ is *possible*
- $AX(\phi)$ ϕ holds in all successor states
- $EX(\phi)$ there is a successor state where ϕ holds
- $A(\phi U \psi)$ along all paths, ϕ is true until ψ becomes true
- $E(\phi U \psi)$ along some path, ϕ is true until ψ becomes true

A temporal formula ϕ is said to hold for FTS M if ϕ is true in the initial state $\hat{s}$. If this is the case, M is said to be a *model* for ϕ.

2.2.2 Correctness specifications

Typical examples of correctness specifications are safety properties, liveness properties, and precedence properties:

- A safety property expresses the notion that "nothing bad will ever happen", by stating that some invariant is true at all times. For example, $AG(x \geq 1)$ asserts that x is greater than or equal to 1 in every state.
- A liveness property states that the truth of one condition will always eventually be followed by the truth of another: $AG(\mathit{trying} \Rightarrow AF(\mathit{critical}))$ asserts that wherever *trying* holds, *critical* will also eventually become true. This is often used to specify that every request will be met with a response, or that every message sent will be met with a reply.
- Precedence properties state that the truth of one property will be preceded by a period of continuous truth for another: $A(\mathit{overflow}\, U \mathit{reset})$ asserts that *overflow* remains true until the moment that *reset* becomes true.

2.3 Model checking algorithms

The last two sections describe a formalism for the modelling of system behaviour (FTSs) and a formalism for the specification of correct behaviour (CTL). With this background, it is possible to give an exact formal definition of the model checking problem: given as input an FTS M and a temporal logic formula ϕ, is M a model for ϕ?

This question can be refined even further. The *local* model checking problem asks whether a particular state, usually the initial state of a system, satisfies the correctness property, whereas the *global* model checking problem aims to determine all states of the system that satisfy the property. The global version of the problem clearly subsumes the local version.

The first algorithms to decide the model checking problem were developed independently in the early 1980s by Clarke and Emerson [10] and by Queille and Sifakis [47]. These algorithms build the entire state graph beforehand and then, starting with atomic propositions and gradually

examining longer subformulas of the correctness specification, iterate over the state graph until finally every state is marked with all subformulas of the correctness property that are satisfied by that particular state.

Although the running time of the algorithm is $O((|S| + |R|) \times |\phi|)$, where $|S|$ is the number of states in the state graph, $|R|$ is the number of transitions, and $|\phi|$ is the length of the correctness property, this approach does not fare well in practise, since it is limited by the amount of memory needed to store the state graph and the labelling information.

This problem can be overcome by switching to an *on-the-fly* model checking algorithm. Instead of computing the entire state graph beforehand, it is generated on-the-fly as it is being explored. Only those parts of the state graph that are needed to check the property are generated. In general, errors are found much earlier and the analysis can terminate as soon as an error is found. Very different algorithms are required for on-the-fly model checking. Such algorithms can be classified as one of two types: automata-based and structure-based.

2.3.1 Automata-based algorithms

Regular finite automata that recognise a language of only finite words can easily be extended to Ω-*automata* that can recognise infinite words. These automata make it possible to translate a temporal logic formula to an equivalent automaton that recognises exactly those infinite sequences of states that satisfy the formula. Figure 2.3 shows an automaton that accepts paths that satisfy $AG(t \Rightarrow AF(c))$. Only paths that visit the state q_0 infinitely often are accepted.

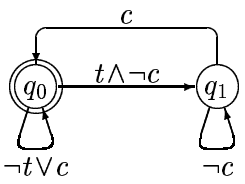


Figure 2.3: Automaton that accepts $AG(t \Rightarrow AF(c))$

Using this knowledge, the model checking problem can be approached in a new way: given a correctness property ϕ, its negation $\neg\phi$ is encoded as an Ω-automaton $M_{\neg\phi}$. By computing

the product of $M_{\neg\phi}$ and an FTS M and checking that this product is empty we can determine whether M satisfies ϕ. If the product is not empty the FTS contains at least one path that violates ϕ. The first automata-based model checking algorithms were developed for LTL by Vardi and Wolper in 1986 [55]; other algorithms for LTL appear in [3, 12], an algorithm for CTL appears in [4], and for CTL* in [5].

2.3.2 Structure-based algorithms

Several algorithms have been developed that direct the exploration of the state graph based on the structure of the temporal logic formula itself. Such structure-based algorithms (also known as *subgoaling* or *induction-based model checking*) make use of the inductive definitions of CTL operators. For example, when checking $AG(\phi)$ such an algorithm would make use of the fact that $AG(\phi) = \phi \wedge AX(AG(\phi))$. It suffices to check that ϕ holds in the current state, and to then explore all successor states (because of the AX operator) and check that $AG(\phi)$ holds in each of them.

Another example illustrates how this approach can avoid unnecessary work: to check $AG(\phi \Rightarrow \psi)$ the algorithm explores the state graph to find states where ϕ holds. Only in those states is the formula ψ investigated.

Structure-based algorithms have been developed for CTL [20, 56], and CTL* [5].

2.3.3 Avoiding redundant work

In practice, the majority of states is reachable from the initial state via more than one path. It is therefore possible that the same state is encountered more than once during the on-the-fly exploration of the state graph. To avoid unnecessarily re-exploring states the state generator must store as many states as possible in the available memory. This task dominates the memory requirements of most on-the-fly model checkers.

2.4 The state explosion problem

The number of states in almost any system of interest, is huge. The size of the state space of a system grows exponentially with the number of its processes and variables. This phenomenon is known as the *state explosion problem*. At first sight, the state explosion problem appears so formidable that model checking of practical systems seems to be hopeless. However, this problem has been studied extensively in the literature—a recent survey is [53].

The major source of state explosion is the interleaving of the concurrent actions of component processes. In the worst case, a system with n non-interacting processes each with k local states has a total of k^n global states. This state space contains all possible orderings of the actions. However, many interleavings are equivalent as far as model checking the correctness specification is concerned and by selecting a single, representative interleaving and ignoring all other interleavings the amount of work required can be reduced. This idea has led to the development of a series of techniques starting with *stubborn sets* [51, 52] in 1988 and including *persistent sets* [24] and *ample sets* [44]. These techniques are widely known as *partial order methods*.

The basis of these techniques is to define conditions under which some enabled transitions may be safely ignored while preserving the property under investigation. Figure 2.4 shows the reduced state graph of the mutual exclusion problem to preserve the property $AG(t \Rightarrow AF(c))$ for the "left-hand" process (cf. Figure 2.2). This reduction is based on the technique described in [23]. In each of the reduced states, indicated by the darker circles, only one of all possible transitions was retained. One of eight states and five of fourteen transitions have been eliminated.

For large state graphs partial order techniques can lead to dramatic increases in both runtime and memory efficiency. Although these techniques are useful for alleviating the state explosion problem, they are, due to the limited scope of this thesis, not discussed further.

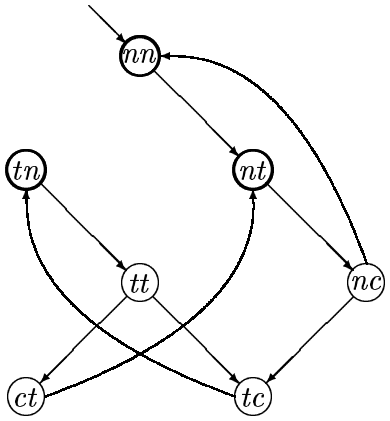


Figure 2.4: Reduced state graph of the mutual exclusion problem

2.5 Fairness

A last but important aspect of model checking is illustrated by the state graph in Figure 2.5. Does the formula $AF(p)$ hold for this system? In other words, do all paths starting at s_0 eventually reach a state where p is true? Apparently there is a valid, infinite execution path $\sigma_u = s_0, s_1, s_0, s_1, s_0, \ldots$ that violates the formula, since p is false in both s_0 and s_1 and therefore false in every state along this path. Even though the transition t_2 can make p true, and is infinitely often enabled along σ_u, it is never executed. Does it make sense to ignore the transition in this way?

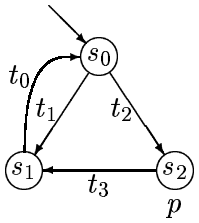


Figure 2.5: State graph with fairness problems

Even though it is the task of a model checker to investigate every possible execution path, the behaviour typified by σ_u is often undesirable: it is in some sense *unfair*, and should be excluded to obtain a more reasonable model of the system.

Consider another example. A system consists of two processes: R, representing a reactive system, and E, its environment. Although they interact from time to time, each process should be allowed to execute internal actions. However, unless the system specifically disallows it, one possible execution path could contain an infinite number of transitions of E without ever giving R the opportunity to progress. This is clearly unacceptable.

The kind of restriction placed on the set of accepted execution paths is called *fairness constraints* [21]. In [19] three main forms of fairness are identified:

- *Impartiality* (or *unconditional fairness*): a path is impartial if every process is executed infinitely often along the path.
- *Weak fairness*: a path is weakly fair if every transition that is enabled almost everywhere (in other words, in every state of the path from a certain state onwards), is executed infinitely often.
- *Strong fairness*: a path is strongly fair if every transition that is enabled infinitely often, is executed infinitely often.

It is obvious that strong fairness subsumes weak fairness, while impartiality subsumes both strong and weak fairness. In practice, however, impartiality usually eliminates many paths of interest.

Fairness is a desirable property for a model checker to support. Not all temporal logic formulas are affected by fairness. For example, safety properties are valid (or invalid) independently of fairness constraints. On the other hand, liveness and precedence properties are, in general, difficult to model check without specifying fairness constraints.

Model checkers without support for fairness would report numerous errors similar to σ_u above. The user of such a model checker has two choices. Firstly, it is possible to explicitly add fairness constraints to the correctness specification in the form of extra clauses. In the above case the specification $AG(AF(t_2)) \Rightarrow AF(p)$ would check the property and at the same time ensure impartiality towards the process containing transition t_2. It would in fact be more desirable to encode strong fairness in this example, but unfortunately CTL is limited to impartiality and weak fairness [18]. LTL, on the other hand, can express all three forms of fairness.

The second option is to encode restrictions in the model itself. For instance, in the example above a counter could be introduced to ensure that the first transition is chosen only a fixed number of times before the model checker is forced to explore transition t_2.

Model checkers with "native" (built-in) support for fairness automatically ignore all unfair paths and allow users to concentrate on the essentials of the model. On the other hand, the danger exists that the correctness of a model will rely on fairness constraints that are not available in a realistic execution environment; users should be made aware of such assumptions.

Chapter 3

Design issues

The design of a model checker starts with the selection of a model checking algorithm. This entails a choice between LTL and CTL or perhaps some other temporal logic, between local and global algorithms, and between structure-based and automata-based algorithms.

Once an algorithm has been selected, it can usually be implemented in a few hundred lines of code. The focus then turns to the on-the-fly generation of states, and here the objectives are simple:

- explore as many states as possible, and
- explore the states as fast as possible.

Another important design decision is the choice of language in which models are expressed. Finite transition systems do not provide adequate abstractions required by users to describe complex models. Instead, systems are usually described in a more expressive, higher-level specification language. The semantics of the specification language dictate to some extent the design of the state generator, so it is important to study the issues before selecting or designing a language.

Guiding principles

Apart from the stated objectives, there are some overriding, implicit goals such as ensuring that state exploration is performed correctly and that the implementation is easy to maintain. The following guidelines, although obvious, are invaluable:

- *Prefer simple mechanisms to complex ones.* Often a designer faces the option of implementing a complex mechanism to reduce storage or runtime requirements. Sometimes such a mechanism can lead to a dramatic increase in performance (as is the case with symbolic model checking [43]), but more often the ultimate effect of an idea is not entirely clear. All other things being equal, simple mechanisms are easier to implement correctly and should be preferred to more complex schemes.

- *Space efficiency trumps time efficiency.* In many cases a trade-off between space and time efficiency is possible. Space efficiency is more important than runtime efficiency, since it is usually more acceptable to wait longer for a result, than to find that the analysis cannot be completed because the model checker has run out of memory.

- *Keep the design structured.* The different tasks of the model checker should be relegated to different components that interact only through well-defined interfaces. Although skeptics claim that a structured design removes opportunities for optimisations, this approach has proven itself time and again. Our experience has been that, even though a model checker may not be large in terms of lines of code, the different algorithms and their interaction can be exceedingly complex. The separation of concerns afforded by modularisation and encapsulation can aid the development of a reliable model checker greatly.

3.1 Requirements

To a model checking algorithm the rest of a model checker is simply an engine that explores the state graph on-the-fly in depth-first order. The model checking algorithm guides the exploration with the following three basic functions:

Execute(): Generate the next state from the current state

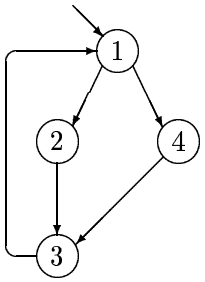
Backtrack(): Fall back one state

Evaluate(p): Evaluate atomic proposition p in the current state

Procedure *Backtrack* does not return a value, and *Evaluate* returns either true or false to indicate the value of the atomic proposition. The *Execute* routine can return the following values depending on the structure of the state graph:

- **Forward:** a new state has been generated and has become the current state.
- **Revisit:** a state was generated but it has been explored before. The current state remains unchanged.
- **Loop:** a state was generated but it forms a cycle. The current state remains unchanged.
- **AllChildrenExplored:** all the children of the current state have been explored. The current state remains unchanged.
- **Complete:** similar to *AllChildrenExplored*. In addition, the current state is the root (initial state) of the state graph.

Figure 3.1 illustrates the interaction between the model checking algorithm and the state generator during the exhaustive exploration of a simple state graph. The initial state is state 1. Each row of the table shows the current state, the routine called by the model checking algorithm, the resulting state (the new current state), and the value returned by the routine. As the example shows, a complete exploration of the state graph requires $2|S| - 1 + C$ calls of



Current state	Call	Next state	Return value
1	Execute	2	Forward
2	Execute	3	Forward
3	Execute	3	Loop
3	Execute	3	AllChildrenExplored
3	Backtrack	2	
2	Execute	2	AllChildrenExplored
2	Backtrack	1	
1	Execute	4	Forward
4	Execute	4	Revisit
4	Execute	4	AllChildrenExplored
4	Backtrack	1	
1	Execute	1	Complete

Figure 3.1: Exhaustive depth-first exploration of a simple state graph

Execute and $|S| - 1$ calls of *Backtrack*, where $|S|$ is the number of states in the state graph, and C is the number of loops and revisits.

The functioning of the state exploration engine can be broken into three important tasks:

- the generation of states,
- the detection of revisited states, and
- the efficient internal representation of states.

These tasks are discussed in Sections 3.2, 3.3, and 3.4 respectively. Section 3.5 considers the question of how fairness can be supported efficiently.

3.2 State generation

Specification languages allow the user to describe a system as a set of processes acting on a set of variables. Each process is a sequence of commands that form the transitions of the state graph. Although state generation will be discussed on a general level, it will be convenient to refer to a concrete example from time to time. In this section examples are expressed in the ESML specification language [14]. A full description of this language can be found in Appendix D, although the examples will be simple enough so make the meaning clear.

To explore a state graph the state generator must keep track of the following information:

- *the current state* that records the the current value of the variables of the system; since the state is not just a single value, but rather the values of a set of variables, this data structure is known as the *state vector*;
- *the depth-first stack* that stores the states on the current execution path and the last transition explored in each state;
- *the transition table* that encodes all possible transitions;
- *the activation list* that stores information about the processes that are active in the current state; and
- *the state store* that records all unique states to detect when states are revisited.

When *Execute* is invoked, it searches for an enabled transition to explore. To avoid re-exploring a transition that has already been tried it uses the information about the last transition stored in the depth-first stack. When a suitable transition has been found, it is executed to generate a potentially new state. This state is checked to see whether it appears on the stack (*Loop*) or in the state store (*Revisit*). If no transitions can be found to explore, *Execute* returns *AllChildrenExplored* or *Complete*.

3.2.1 The execution of transitions

Each transition of the state graph can be viewed as a guarded command pair *guard* $\rightarrow$ *action*. The *guard* checks that the transition is enabled, and the *action* is a set of assignments (changes to the current state) that effect the transition. For example, the assignment `n := n + 1` can be viewed as the guarded command

$$(loc = 4) \rightarrow n := n + 1;\ loc := 5$$

The location counter of the process that contains the assignment is stored in the *loc* variable. The guard is satisfied when the location of the assignment is reached (location 4 in this example). The action effects the assignment and updates the location counter.

Pre-compiled transition systems

One way to encode such transitions is to parse the specification of the system and to translate it to equivalent code in a suitable programming language. The code is then compiled and linked with the rest of the model checker to form an executable image. This approach is used in the SPIN system to translate Promela models to C code [34], and in a previous model checker developed at the University of Stellenbosch to translate ESML models to Modula-2 code [58].

The translator generates one procedure for each process in the specification. This procedure contains the actions for the transitions of the corresponding process. An additional procedure is generated to initialise the transition table with the correct values. For example, consider the following ESML process definition:

```
1  PROCESS Counter;
2  VAR n: int;
3  BEGIN
4     n := 0;
5     DO n<10 -> n := n + 1
6     [] n=10 -> n := 0
7     END
8  END Counter;
```

The process initialises the value of `n` to 0. The `DO` command has the same semantics as the repetition construct in Dijkstra's guarded command language [16]: it executes while one or more of the guards are true. If `n` is less than 10, it is incremented; if `n` is 10, its value is reset to 0.

The generated code and the appropriate fragment of the transition table is shown in Figure 3.2. The first field *Process* of the transition table identifies the process to which the transition belongs (in the example the `Counter` process's number was arbitrarily chosen to be 2). The *Number* field stores the transition number, the *NextTransition* field stores the location of the next transition, and the *Action* field identifies the corresponding action of the transition. Each guard of the `DO` construct is encoded as a separate transition. The *NextGuard* field of the transition table stores the location of the next guard to be executed in case the current transition fails.

Additional code is generated to initialise the transition table:

```
1  PROCEDURE InitTransitionTable;
2  BEGIN
3     ...
4     MakeTrans(2, 0, 1, 1, empty);
5     MakeTrans(2, 1, 2, 2, empty);
6     MakeTrans(2, 2, 1, 3, empty);
7     MakeTrans(2, 3, 4, 4, empty);
8     MakeTrans(2, 4, 1, 5, empty);
9     MakeTrans(2, 5, 6, empty, empty);
10    MakeTrans(2, 6, empty, 6, empty);
11    ...
12 END InitTransitionTable;
```

When *Execute* has selected a process from the activation list, it retrieves the process's location counter from the state vector. The location counter is the number of the transition the process is about to execute. It looks up the transition in the transition table and calls the corresponding process procedure (such as `CounterProcess`) to which it passes the current state vector and the value of the *Action* field for the selected transition. The `CASE` in procedure `CounterProcess` selects the appropriate action to execute. When the procedure returns, *Execute* checks whether a new state has been generated and acts accordingly.

Process	Number	NextTransition	Action	NextGuard	
2	0	1	1	-	`n := 0`
2	1	2	2	3	`n < 0`
2	2	1	3	-	`n := n + 1`
2	3	4	4	5	`n = 10`
2	4	1	5	-	`n := 0`
2	5	6	-	DO	`END`
2	6	-	6	-	`END Counter`

```
1  PROCEDURE CounterProcess(VAR state: StateVector; action: INTEGER);
2  VAR expr: INTEGER;
3  BEGIN
4     CASE action OF
5       1: IF TRUE THEN
6             SetVar(state, pos_n, 0);
7             SetVar(state, pos_loc, 1)
8          END
9     | 2: IF GetVar(state, pos_n) < 10 THEN
10            SetVar(state, pos_loc, 2)
11         END
12    | 3: IF TRUE THEN
13            expr := GetVar(state, pos_n) + 1;
14            SetVar(state, pos_n, expr);
15            SetVar(state, pos_loc, 1)
16         END
17    | 4: IF GetVar(state, pos_n) = 10 THEN
18            SetVar(state, pos_loc, 4)
19         END
20    | 5: IF TRUE THEN
21            SetVar(state, pos_n, 0);
22            SetVar(state, pos_loc, 1)
23         END
24    | 6: (* Remove the Counter process from the activation list *)
25    END
26 END CounterProcess;
```

Figure 3.2: The transition table and code generated for process `Counter`

Our experience with a model checker that uses this approach has taught us that it is complex and error-prone. The generated code is difficult to read and relate to the original model. A frequent problem was that correcting the code generation for one model, caused the translator to generate erroneous code for another. A part of the complexity of this approach stems from the fact that the semantics of the specification language must be encoded almost entirely in the generated transition system.

Interpreted transition systems

An alternative to pre-compiling the transition system is to encode each action as a set of instructions that are interpreted to execute the transition.

Since the mid seventies, designers of compilers have advocated specialised abstract machines for high-level languages [6, 38, 49, 60, 61]. The use of an interpreter has several advantages: code generation is simpler when the target instruction set is specifically designed for the high-level language in question, and it is easy to check that the correct instructions are generated. Interpreters also offer greater security and portability than compilers. Moreover, an interpreter consists of instructions that can be tested separately—a very attractive feature for the generation of states.

The abstract code for the `Counter` process is shown in Figure 3.3. The target machine in this case is a straight-forward stack-based abstract machine. When abstract code is generated, the transition table is not needed. The location counter of each process stores the address of the next instruction it is about to execute. The abstract interpreter decodes and executes the code until it reaches an instruction that triggers a transition; then *Execute* examines the new states and handles it appropriately.

The instructions in Figure 3.3 are interpreted as follows: `pushValue 0` pushes the constant 0 on the expression stack, and `storeVariable 1` removes the value and stores it at variable address 1 (the address of `n`). The `guard 120` instruction signals to the interpreter that the following few instructions are the guard of a `DO` command and that, should this guard fail, the next guard can be found at address 120. The `pushVariable 1` instruction at address 106 places the value of `n` on the stack and `pushValue 10` places 10 on the stack. The `ifless` instruction removes

```
100 pushValue 0              n := 0
102 storeVariable 1
104 guard 120                DO
106 pushVariable 1           n < 10 ->
108 pushValue 10
110 ifless
111 pushVariable 1           n := n + 1
113 pushValue 1
115 add
116 storeVariable 1
118 jump 104                 jump to start of DO
120 guard 133                []
122 pushVariable 1           n = 10 ->
124 pushValue 10
126 ifequal
127 pushValue 0              n := 0
129 storeVariable 1
131 jump 104                 jump to start of DO
133 endguards                END DO
134 terminate                END Counter
```

Figure 3.3: Abstract code for process `Counter`

the top two elements of the stack and checks whether they satisfy the less-than relation. If so, a transition has been completed and the location counter is advanced to the next address, 111. Otherwise, the interpreter will execute the next guard at address 120. The instructions at addresses 111–116 encode another assignment, and the `jump 104` at address 118 directs the flow of control back to the beginning of the `DO`—this command will also trigger a transition to prevent the interpreter from getting stuck in a non-terminating cycle. The instructions for the second guard (addresses 120–131) are similar. The `endguards` instruction indicates that there are no more guards to evaluate. If none of the guards evaluated to true and the `endguards` instruction is reached, the interpreter advances the location counter to the next address, since the `DO` has then terminated. Lastly, the `terminate` instruction terminates the executing process.

Although it is generally accepted that interpretation of programming languages is slower than executing native machine instructions, the same is not necessarily true for model checking. The

actions of transitions involve complex operations and there are several other tasks that need to be handled during model checking. For these reasons, the overhead of interpretation may turn out to be negligible. Furthermore, the complexity of the action code shown in Figure 3.2 is distributed among different abstract instructions. The implementation of each instruction is independent of that of others. In contrast to the generated transition system, the abstract code is easy to read when one is familiar with the instructions.

Non-determinism, communication, and dynamic process creation

Figure 3.4 shows an ESML model of the mutual exclusion problem. The model defines two processes, `Semaphore` and `User` that communicate via the global communication channel `s`. The channel is of type `sema`, which can convey two messages, `P` and `V`. `Semaphore` manages a Boolean semaphore called `free`. When it receives a `P` request and the semaphore is available, it is granted and set to `FALSE`. Upon receiving a `V` request the semaphore is released. The `User` process can either stay in its noncritical region or cycle through its noncritical, trying and critical regions. This choice is made non-deterministically by the `DO` command in line 23. Before the `User` process enters the critical region it requests the semaphore, which it releases again after leaving the critical region.

This model illustrates the use of three kinds of transitions that require special attention: non-deterministic choice (lines 23 and 24), communication (lines 13, 14, 24, and 26), and process creation (lines 31–32). These issues are addressed in the sections below.

3.2.2 Non-determinism

The state generator must ensure that all non-deterministic choices are explored. In general, non-determinism is not difficult to implement but care must be taken to handle all cases correctly.

In the case of a pre-compiled transition system this means that the *NextGuard* field is used to evaluate all guards of a `DO` construct. When falling back, *Execute* examines the *NextGuard* field of the last transition to execute to determine whether there are other guards to investigate.

```
1  MODEL ME;
2  TYPE
3    region = noncrit, try, crit;
4    sema = {P, V};
5  VAR
6    s: sema;
7
8  PROCESS Semaphore(IN s: sema);
9  VAR free: BOOLEAN;
10 BEGIN
11   free := TRUE;
12   DO TRUE ->
13     POLL s?P & free -> free := FALSE
14       [] s?V        -> free := TRUE
15     END
16   END
17 END Semaphore;
18
19 PROCESS User(OUT s: sema);
20 VAR r: region;
21 BEGIN
22   r := noncrit;
23   DO TRUE -> SKIP
24   [] TRUE -> r := try; s!P;
25              r := crit;
26              r := noncrit; s!V
27   END
28 END User;
29
30 BEGIN
31   User(s); User(s);
32   Semaphore(s)
33 END ME
```

Figure 3.4: ESML model of the mutual exclusion problem

The abstract machine interpreter must also cater for this situation. If falling back, the last instruction to execute was a `guard` instruction, the address of the next guard is fetched and it is executed.

3.2.3 Communication

Processes can communicate with each other in two ways: either through shared variables, or through message passing. Shared variables are simpler: apart from assignments, no other operations or data structures are required. However, message passing is often a more accurate and convenient way of describing process interaction. Message passing entails communication channels between processes, the messages themselves, and `SEND` and `RECEIVE` operations to dispatch outgoing and accept incoming messages. Two forms of message passing are commonly used: asynchronous and synchronous.

Asynchronous communication

Asynchronous communication makes use of communication queues to pass messages between processes. The `SEND` operation is always enabled as long as the communication queue is not full; the sender's message is inserted into the queue immediately and the process can continue execution. Similarly, `RECEIVE`s are enabled as long as there are waiting messages in the queue. The semantics of the specification language must define what happens when the queue is either full or empty. The operation can either fail, or block until a message or open slot becomes available. In the SPIN system the user can specify the behaviour of operations under these conditions [30].

The communication queues must form part of the state vector: a state where a message is waiting in a queue is clearly different from a state where the queue is empty. To avoid arbitrarily large state vectors, communication queues are bounded in length. A model is only correct under the assumption of specific queue bounds. A similar model with shorter or longer queues does not necessarily satisfy the same correctness property. The bounds that can be model checked are usually much smaller than those of actual systems: an accepted and usually valid simplification is that if a model is correct for queues of a certain length is that it will also be correct for longer

queues. The user is obliged to find a minimum, hopefully representative, configuration of a system.

Asynchronous `SEND` operations carry the same cost as two variable assignments, but `RECEIVE`s are somewhat more expensive, since the head message must be copied to the process variables, the queue contents must be shifted along one slot, and the queue length must be decremented. If there are n messages waiting in a queue, a `RECEIVE` is equivalent to $n+2$ assignments. Storing these queues in as circular buffers is not viable, since the same queue contents can be stored shifted at different offsets in the buffer, causing a considerable increase in the number of states.

Synchronous communication

In the case of synchronous message passing both `SEND` and `RECEIVE` operations block and are not enabled until a suitable partner becomes available. At this point the processes synchronise: both operations become enabled and can execute simultaneously in a single transition. Messages are never stored "between processes" but are instantaneously moved from the sender process to the receiver process. This form of communication is popular since it is generally accepted that synchronous communication is simpler to use and implement.

It is often useful for a receiver process not to block on a single message, but to be ready to send and receive one of several messages. This need is addressed by a selective receive operation such as the `POLL` command of the ESML specification language [14]. It consists of one or more guarded commands: the guards in this case are communication operations. The `POLL` command blocks until one or more guards are enabled. It then non-deterministically selects one of the enabled guards, and executes its corresponding action. For example, the following `POLL` command blocks until it can send message `a` on channel `ch0`, or receive on channels `ch1` or `ch2`.

```
1 POLL
2    SEND(ch0, a)    -> A
3 [] RECEIVE(ch1, b) -> B
4 [] RECEIVE(ch2, b) -> C
5 END
```

When a communication operation is reached, the state generator must determine whether a synchronisation partner is immediately available, or whether the communicating process must block while it waits for a matching operation. One way of storing information about the availability of partners is by using *channel queues*: each channel has a corresponding queue where waiting partners are stored as they arise. Only process identification numbers are stored in this queue. A communication operation checks the channel queue for a partner, which it removes from the queue. If no partner is available, the communicating process joins the queue and cannot proceed until a partner arrives to remove it. At any point the queue will contain only sending or only receiving partners. If more than one synchronisation partner is available, the state generator must explore all possible synchronisations.

The `POLL` operation introduces several complications: several channel queues may have to be checked (one for each guard) and, if no partners are available, the process must join all these queues. When another command synchronises with a `POLL` guard, the polling process is removed from all queues it has joined, since once one `POLL` guard is selected, the other guards are disabled. Furthermore, enough information must be stored on the stack so that channel queues can be restored to their previous values when backtracking. These problems escalate when one `POLL` operation is allowed to synchronise with another `POLL`.

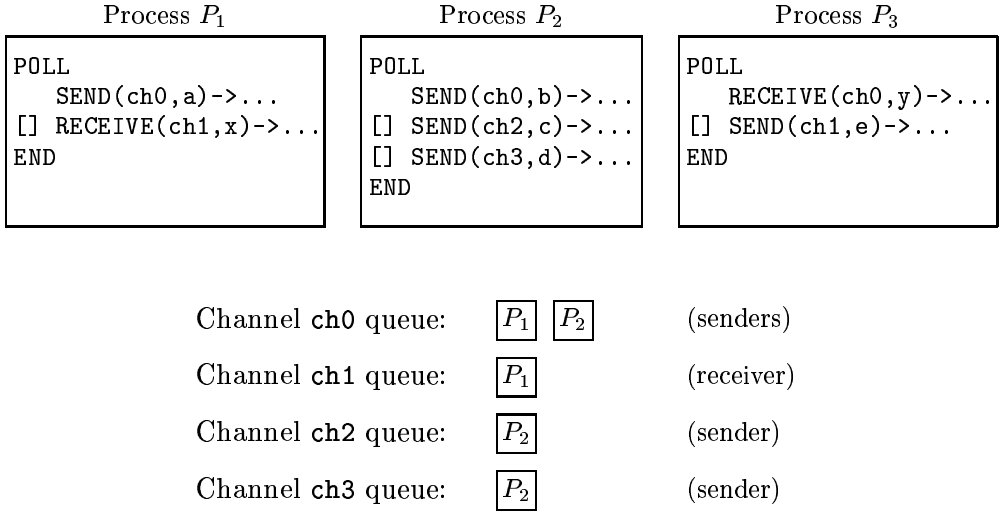


Figure 3.5: Interaction of channel queues and `POLL` commands

As an example, consider the scenario shown in Figure 3.5. Process P_1 and P_2 have reached their

`POLL` commands in that order. In both cases no synchronisation partners were available and the processes were inserted in the appropriate channel queues. When process P_3 reaches it `POLL` command, the state generator must check channel queues `ch0` and `ch1` (the channels addressed by its guard statements). For the first guard two channels are available: if P_3 synchronises with P_1, process P_1 must be removed from channel queues `ch0` and `ch1`; if it synchronises with P_2, process P_1 must be re moved from channel queues `ch0`, `ch2` and `ch3`. The second guard can synchronise only with P_2 in which case process P_2 is once again removed from channel queues `ch0`, `ch2` and `ch3`. In total, process P_3 can synchronise in three ways, yielding three transitions from the state. If the last guard of process P_2 addressed channel `ch2` instead of `ch3`, P_2 would appear twice in channel queue `ch2`, allowing P_3 to synchronise with it in two ways, and resulting in four transitions.

It is not necessary to store channel queues as part of the state vector, since the same information is already available in the location counters of the processes (which can be found in the state vector). In fact, it is possible to implement synchronous communication *without* channel queues in the following way: when a communication operation tries to execute, it scans through the activation list to find other processes that are ready to perform a matching operation. It does this by examining the location counters and next transitions of processes. If a synchronisation partner is found, both processes execute a combined transition. Otherwise, the communication operation remains disabled.

When channel queues are used to support synchronous communication, `SEND` and `RECEIVE` operations carry the same cost as assignments; the `POLL` operation requires more overhead. Very little space is required to store channels: if P is the set of processes and $chan(p)$ is the maximum number of guards in any of its `POLL` commands, 1 if it contains only `SEND` and `RECEIVE` commands, or 0 if it contains no communication operations, the maximum number of processes that can be present in the channel queues is

$$\sum_{p \in P} chan(p)$$

Each channel queue ch must cater for $\max\{chan_{\mathtt{S}}(ch), chan_{\mathtt{R}}(ch)\}$ entries, where $chan_{\mathtt{S}}(ch)$ is the total number of `SEND`s that address channel ch, and $chan_{\mathtt{R}}(ch)$ is the total number of `RECEIVE`s that address channel ch, counted throughout the model.

When scanning is used for synchronous communication no extra storage space is required. Each communication operation has to check every other process, but scanning through the activation list is just as fast as scanning through channel queues.

3.2.4 Dynamic process creation

Some specification languages allow for the creation of new processes during the exploration of the state graph. It is therefore possible that states along different branches of the state graph can contain different processes. When a new process is created its variables are added to the state vector, which must grow to accommodate the additional variables. The state vector must also store information about the number of active processes as well as their order of activation. Unless this *signature* information is present, it is possible that states containing different processes are mistaken for each other.

Furthermore, it is possible that equivalent states cannot be recognised as such. Consider the case where a non-deterministic choice is made to either activate process P_1 and then process P_2, or to activate P_2 and then P_1. As far as the model checking algorithm is concerned, the resulting states are equivalent, but since their process activation order is different, this equivalence cannot be recognised. Despite its artificial nature, this example represents a real problem that could arise in complex models.

Usually the state vector cannot grow arbitrarily large and some upper limit is placed on the number and size of processes. When a process terminates, its portion of the state vector must be reclaimed, or must remain unused during the rest of the analysis. The former option requires that the state vector be rearranged, while the latter results in wasted bits in the state vector. Transitions that create or destroy processes also modify the activation list. These modifications must be undone when the state generator backtracks.

These complications must be weighed against the usefulness of dynamic process creation. A study of ESML and Promela specifications found no models that depend on this feature, or cannot be trivially modified to avoid it. Consequently, this feature has been eliminated from ESML.

Dynamic object creation

To our knowledge there are no specification languages that allow the dynamic creation of data objects. This feature leads to the same problems caused by dynamic process creation, but it is not as easily dismissed. As the power of model checkers increase it is inevitable that users will require this feature. (Ideas on extending SPIN in this way can be found in [15].) Moreover, dynamic objects usually have a high turnover, thus exacerbating the problems described.

3.3 The detection of revisited states

To avoid the redundant effort of re-exploring large parts of the state space, the system must be able to detect when a state is reached a second time, or being "revisited". To accomplish this, as many as possible of the reached states are stored in memory and each new state is checked to establish whether it is really a new state, or whether is has been generated before.

The state store has the following interface:

Insert(s): Insert state s into the state store.

Lookup(s): Check whether state s is present in the state store.

It is tempting to consider the use of dynamic data structures such as binary search trees. Apart from the storage overhead of storing pointers and the runtime overhead of memory management, operations on such complex data structures can be expensive. Usually the sequence of states passed to the *Insert* routine resemble each other closely, leading to unbalanced trees. Although operations for balanced trees cost $O(\log n)$ where n is the number of states stored in the tree, the hidden multiplicative constant in this time bound is usually prohibitively large.

3.3.1 State caching

Most model checkers store states in a table which is accessed by means of closed hashing. Hash collisions are resolved by double hashing to avoid clustering as far as possible. The first slot of

the hash table probed for state s is $h_1(s)$. If s is not found in this position an offset $\delta = h_2(s)$ is calculated and the next slots probed are $h_1(s) + \delta$, then $h_1(s) + 2\delta$, then $h_1(s) + 3\delta$, ...

While the hash table remains relatively empty, hashing provides roughly constant time access to the states. As the table fills up, the number of collisions increase and the access time deteriorates until the table is full and the analysis must be aborted. Fortunately, this can be remedied by using the fact that states in the table may be overwritten. If such a "lost" state is subsequently revisited, the state and its children are re-explored. This does not invalidate the analysis, but simply leads to extra work. Moreover, the children of the revisited but lost state may still be in the table and may prevent the re-exploration of the entire subgraph.

Clearly, the quality of the hash functions is critical to the performance of the state cache. However, it is difficult to exploit knowledge about the nature of the model to produce an intelligent hash function. Because of the high frequency of lookups, it is important that the hash function remains as simple and fast to compute as possible. It is important that no bits of the state vector are ignored by the hash function. Consequently, a function based on low-level bit manipulation operations works well.

Since the state table does not store all visited states, but merely a subset, this technique is known as *state caching* [31]. In addition, the cost of insertion and searching can be controlled by limiting the number of probes. For example, when the limit is exceeded during an *Insert* operation, an older state is immediately replaced. Empirical results show that a cache of states can be effective for state graphs 2–3 times the size of the table.

Holzmann has investigated several strategies for selecting which state to overwrite. These strategies included replacing most frequently visited states, least frequently visited states, smallest subgraph states (states that form the root of the smallest subgraph of the state graph), random states from largest class (a class if formed by all states visited equally often), and random replacement. His results show that random replacement is the best strategy—the probability of revisiting a state is not strongly correlated with the number of previous visits [31].

Unfortunately, this approach and those presented below suffer from poor locality of reference. Sequential operations are unlikely to access the same part of the state table and therefore swapping memory pages to disk does not help but hinders the efficiency of state caching.

3.3.2 Bitstate hashing

An alternative that makes more effective use of memory, is the *bitstate hashing technique* [32, 33]. This technique uses a large fixed-size array of bits to keep track of visited states. When a new state is generated, a hashing function is used to compute an index into the bit array for the value of the state. The corresponding bit is set to indicate that the state has been visited and can be checked to detect when the state is revisited.

Unfortunately, this technique has a serious drawback. While any errors found during the analysis are genuine, bitstate hashing cannot guarantee that the correctness property holds for a model. Hashing conflicts cannot be resolved in the usual way because information about the original state is not stored in the bit array. Two states may therefore map to the same bit without the model being able to detect it. If the system visits the second of the two states, it will erroneously decide that the state has been explored before and that it is being revisited. In this case some parts of the state graph may be ignored.

This problem is ameliorated by the observation that since the size of the bit array is very large, the probability of collisions is exceedingly small. Moreover, collisions do not necessarily result in false positives. Holzmann has suggested the simultaneous use of two bit arrays with statistically independent hash functions [34]. Another approach is to use a single bit array and to rerun the analysis several times, each time with an independent hash function. Wolper and Holzmann have made careful studies of the trade-offs involved in using multiple bit vectors, multiple runs with a single a bit vector, and also other approaches [36, 63].

3.3.3 State caching v. bitstate hashing

When the state vector is large, state caching can handle only relatively small models. On a typical workstation with 64 megabytes of memory, $M = 2^{29}$ bits are available for the state store. If each state vector is stored in $S = 2^{13}$ bits (the default state vector size in SPIN), only $N = M/S = 2^{16}$ or roughly 65000 states can be checked. In this case the bitstate hashing technique is clearly superior, since only trivial models can be checked with a state cache.

However, the techniques discussed in Section 3.4 make it possible to compress the same states to

roughly $S = 2^8$ bits, allowing $N = 2^{21}$ or roughly 2 million states to be checked. Furthermore, the use of partial order techniques makes it possible to efficiently check state graphs of upto roughly 2^5 times the size of the state cache, yielding $N = 2^{26}$ states [26].

Given the capability of the state cache in combination with effective state compression and optionally partial orders, it makes sense to use bitstate hashing only as a last resort, and to prefer state caching for the normal operation of a model checker.

3.3.4 Implicit representations

Several implicit state representation schemes have been suggested. These techniques use specialised graph encodings [27], minimised automata [37] or BDDs [59] to represent the set of reached states. Such techniques can have a dramatic impact on the memory requirements: in [37] results show that in general, memory use is reduced by a factor of 4 and in some cases by a factor of 17. Unfortunately, these gains come with at least a tenfold increase in execution time, and in some cases additional training runs are required to yield results. As noted in [27], due to their high runtime costs, such schemes cannot be used for the normal operation of a model checker when the probability of finding an error is high and the duration of runs is low.

3.4 The representation of states

The representation of states is a critical part of the model checker, since it affects every aspect of its operation. To the rest of the system only the following interface is available:

Compare(s_1, s_2): Check whether two states are equal.

Assign(v, s): Assign a copy of state s to state variable v.

GetValue(s, i): Return the value of variable i in state s.

SetValue(s, i, v): Set the value of variable i in state s to v.

Ideally, these operations should be encapsulated in a module that exports the state vector as an abstract data type to the rest of the model checker. This allows the underlying implementation

to be changed without affecting the rest of the system.

3.4.1 General issues

The routines above must be implemented as efficiently as possible. However, aside from the primary objectives of using as little memory as possible per state, and making the operations as fast as possible, several other considerations should be taken into account.

The relative frequency of operations

One guideline when selecting a representation scheme is the relative frequency of operations: not all operations occur with the same frequency and therefore it makes sense to optimise the commoner operations while allowing less frequent operations to be more expensive.

The *Assign* operation is used to copy state vectors to the stack and the state table, resulting in two calls for each unique state that is explored. *Compare* is used when checking a new state against the states on the stack (for detecting loops) and the states in the state table (for detecting revisits). The frequency of this operation depends on the load of the state table: if the state table is relatively empty, there will, on average, be only one *Compare* instruction for each unique state. If the state is represented explicitly as a string of b bits, the cost of the *Assign* operation is $\Omega(b)$ and that of *Compare* is $O(b)$; if a more complicated, say graph-based, representation is used, these operations may be more expensive.

GetValue operations are used whenever the value of a variable is accessed. Several variables are examined to evaluate the transition guard and the right-hand side of assignments. Each transition will involve at least one *SetValue* operation to update the value of the location counter for the process that executed the transition. Most transitions also change the value of a variable; each change requires further *SetValue* operations.

In practice the following trend emerges: *GetValue* is the most frequent operation, and *Assign* the least frequent. The relative frequency of *SetValue* and *Compare* operations depend on the specific state graph. For large models we found that the number of *Compare*'s dominate.

Fixed v. variable length state vectors

Fixed length state vectors are desirable since this simplifies the implementation of the interface by eliminating checks for special conditions. If states of varying lengths are allowed the state table must be implemented as a table of pointers, rather than an array of states. The use of dynamic memory to allocate storage space for states incurs further overhead on the state manipulation operations. If states are discarded (overwritten in the state table) the problem of memory fragmentation must be addressed in some way, resulting in yet more overhead.

However, it is sometimes necessary to add variables to the state vector during the analysis of a model, as when a new process is created. This problem is overcome by fixing the size of the state vector to some upper approximation of the maximum required size. Although the unused bits of some, perhaps even most, states are wasted, this approach is runtime efficient and usually a reasonably tight approximation is possible. An extra field is needed to describe the active length of each state vector. When variables in the state vector is guaranteed not to be used again, it is possible to reuse the space they occupied, but this operation is generally too expensive to implement and requires that the state vector contains additional information to describe its composition.

Explicit v. implicit storage of variables

Some representation schemes allocate a fixed set of bits to each variable, making it possible to extract the value of variables directly from the state vector. This *explicit* storage of variables stands in contrast to other, *implicit* approaches where variable values are encoded in more complicated ways and where it is not possible to associate a fixed set of bits with each variable. Explicit storage is obviously preferable to implicit encodings that require computation to yield variable values, especially in light of the fact that *GetValue* is the most frequently used operation.

Duplication of state information

Many of the problems surrounding state representation can be resolved by maintaining both a compressed and uncompressed copy of the current state vector. *GetValue* operations fetch values from the uncompressed copy while *SetValue* acts on both data structures. Only compressed state vectors are stored in the state table. It is therefore never necessary to uncompress states and techniques that yield small states quickly can be used, even if the uncompression operation is expensive.

3.4.2 Representation techniques

The simplest approach to the representation of states is to allocate one integer per variable and store each variable in its own slot. Compound variables (such as arrays, records and queues) are stored element by element. This technique requires bn bits of storage, where n is the number of variables and b is the number of bits used to store an integer. The result of encoding the variables of the mutual exclusion ESML model in Figure 3.4 and choosing $b = 16$ will result in a state vector of 96 bits (`LC` is the process location counter):

Semaphore		$\texttt{User}_1$		$\texttt{User}_2$	
`free`	`LC`	`r`	`LC`	`r`	`LC`
95...80	79...64	63...48	47...32	31...16	15...0

Clearly most of the bits in this representation are wasted. For instance, the `free` variable uses only one of the 16 bits it occupies. By allocating only as many bits as is necessary to store the values of a variable, a much smaller state vector is obtained. Assuming that $|\texttt{LC}_{\texttt{Semaphore}}| = 9$ (meaning the the location counter of process `Semaphore` can assume 9 different values), and $|\texttt{LC}_{\texttt{User}}| = 11$, a state vector with 17 bits is obtained:

Semaphore		$\texttt{User}_1$		$\texttt{User}_2$	
`free`	`LC`	`r`	`LC`	`r`	`LC`
16	15...12	11...10	9...6	5...4	3...0

If the number of values variable v_i can assume is $|v_i|$, and there are n variables, the number of bits required by this representation is

$$\sum_{i=1}^{n} \lceil \log_2 |v_i| \rceil$$

For both these techniques the *GetValue* and *SetValue* routines can be implemented efficiently, using bit manipulation in the case of the second technique. The *Assign* and *Compare* operations are expensive for the first, simpler technique since state vectors are larger. Both yield fixed-length state vectors in which variables are represented explicitly.

State enumeration

Optimally, if there are n states in the state graph, each state can be represented in $\lceil \log_2 n \rceil$ bits. Unfortunately, this idea has two flaws: (1) it is generally not known beforehand what the value of n is, and (2) each state must be assigned a unique number in the range $0 \ldots n-1$. There is no natural mapping from state vectors to such numbers, and this idea can only be implemented using a lookup table, that requires storing each full state vector, thus defeating the object of using only $\lceil \log_2 n \rceil$ bits per state.

A variation of this idea is to break states into smaller units, either arbitrarily or on process boundaries [35, 59]. The parts are then enumerated separately with smaller lookup tables and the results are combined to form a smaller state. Figure 3.6 shows how this idea can be applied to the mutual exclusion model. The local states of processes are stored in separate lookup tables. The global state vector shown at the bottom is the combination of indices of local states in the respective lookup tables. In the example, m bits are required where $m = \lceil \log_2 k_{\mathtt{S}} \rceil + \lceil \log_2 k_{\mathtt{U1}} \rceil + \lceil \log_2 k_{\mathtt{U2}} \rceil$.

Hashing can be used to implement local state lookups efficiently. *GetValue* entails extracting the local process state from the state vector and looking up the variable value in the lookup table using the local state as an index. *SetValue* computes the new local process state, performs a lookup to determine the index of the local state, and updates the global state with the computed index. *Assign* and *Compare* operate on the global state in a straightforward manner. The memory needed for lookup tables is usually negligible, compared to the memory required

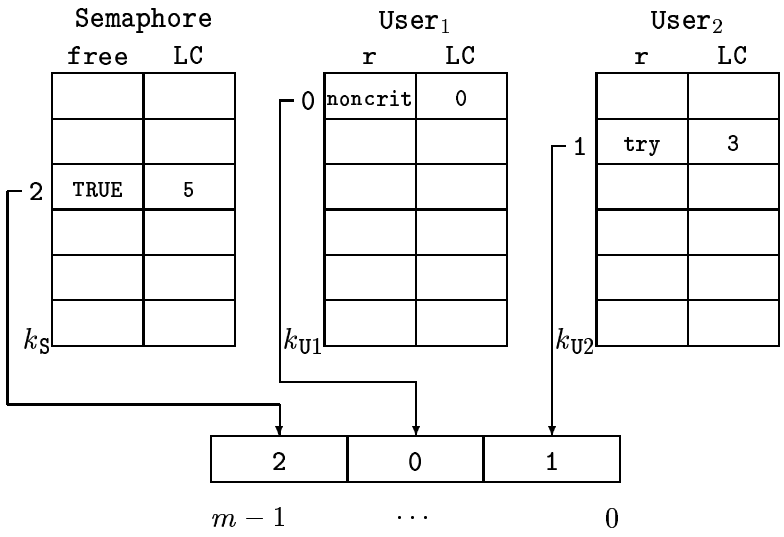


Figure 3.6: Process state enumeration

for the state store. This technique yields fixed-length state vectors in which variables are represented explicitly but indirectly.

The main drawback of the technique is that the analysis must be aborted when any lookup table fills up. For optimal compression the sizes k_i of the lookup tables must be as small as possible while still accommodating all local states. In practice, determining suitable upper approximations for k_i is difficult and is accomplished by performing "training runs" that try to estimate the number of local states. Holzmann has studied this and similar approaches and found that states can be compressed by a factor of 0.27, but only at the expense of a threefold increase in running time, excluding the time taken by training runs [35]. As in the case of bitstate hashing and implicit representations of the state table, this technique is a useful alternative in the last resort, but less suited for the default operation of a model checker.

3.4.3 State compaction

We now present a technique that produces highly compacted, fixed length states that makes it possible to update individual variables without recompacting the state [22, 58(Section 5.1)]. Most variables in validation models range over only a few values and a significant reduction in

state size is possible by simply placing tighter bounds on the ranges of variables and packing them into the minimum space required. Users can easily supply the information needed to do this if the validation language supports user-definable types. For example, type definitions such as `ProcNumber = 0..4` are easy to use and provide enough information to store variables in compacted form.

A small example will illustrate the basic idea. Assume that a model contains three variables v_1, v_2, and v_3 which can respectively assume values from the ranges $0 \ldots 4$, $0 \ldots 2$, and $0 \ldots 6$. The compacted form V of each given state is computed as

$$\begin{aligned} V &= v_3 + 7(v_2 + 3v_1) \\ &= v_3 + 7v_2 + 7 \cdot 3v_1 \end{aligned}$$

Each variable can be thought of as a digit in a variable radix representation of V. It is clear from the first line that "digit" v_3 can range over its seven values $0 \ldots 6$ without affecting the other variables. Similarly, the other variables can range over their respective values without influencing v_3. Two constant factors are associated with each variable v_i. These factors, known as the lower and upper factors of each variable, are denoted by v_i^l and v_i^u respectively. In the example above, $v_3^l = 1$, $v_3^u = 7$, $v_2^l = 7$, $v_2^u = 7 \cdot 3 = 21$, $v_1^l = 21$, and $v_1^u = 7 \cdot 3 \cdot 5 = 105$. These factors are used as masks to extract and update the value of a specific variable in the compacted representation of a state.

A state can now be encoded as a single large integer V. The interface is implemented as follows:

Compare(V_1, V_2): Check whether $V_1 = V_2$.

Assign(V, V_s): Assign $V \leftarrow V_s$.

GetValue(V, i): To obtain the value of variable v_i the higher factor is used to strip out all variables to the right of v_i and the lower factor is used to strip out all variables to the left of v_i.

$$v_i = (V \bmod v_i^h) \operatorname{div} v_i^l$$

SetValue(V, i, v_i'): To change the value of a variable, if the value of v_i changes to v_i', the updated state vector is

$$\begin{aligned} V' &= V - v_i^l \cdot v_i + v_i^l \cdot v_i' \\ &= V + v_i^l \cdot (v_i' - v_i) \end{aligned}$$

The cost of a *GetValue* operation is two multiplications, and that of a *SetValue* operation is two additions and one multiplication. These are the only runtime costs associated with these operations. The costs of *Compare* and *Assign* depend on the length of the state vector.

Assume that the number of values allowed for variable v_i is denoted by $|v_i|$ and that the lower and upper factors associated with v_i are denoted by v_i^l and v_i^u, respectively. The lower factor of variable v_1 is 1 and its upper factor is $|v_1|$. For $i > 1$ the lower and upper factors of variable v_i are given by

$$\begin{aligned} v_i^l &= v_{i-1}^u \\ v_i^u &= |v_i| \cdot v_i^l \end{aligned}$$

If the form of the state vector remains fixed, the computation of the lower and upper factors can occur during the initialisation of the system. However, the computation is simple enough to be performed during the execution of the model—this approach was implemented in [22].

The number of bits required to store a compacted state with n variables is

$$\left\lceil \log_2 \prod_{i=1}^{n} |v_i| \right\rceil$$

For instance, the number of bits required to store v_1, v_2, and v_3 in the example above is $\lceil \log_2 5 \cdot 3 \cdot 7 \rceil = 7$. If state compaction is applied to the mutual exclusion model, the state vector can be stored in 15 bits.

This scheme recommends itself in many ways: operations are not expensive either in terms of time or space; variables are represented explicitly in that their values can easily be extracted from a compacted state; it does not lead to variable length encoding although the active part of a state vector can grow and shrink easily (although it may require the one-time computation

of lower and upper factors for the new variables); and the scheme is optimal in the sense that no explicit encoding can use fewer bits.

3.5 Fairness

Section 2.5 briefly outlined the basic ideas involved in fairness and introduced the three major forms: impartiality, weak fairness and strong fairness. As noted, fairness constraints can either be added to the correctness specification or a model checker can offer intrinsic support for some form of fairness. The former option has no influence on the state generator; this section investigates the latter option.

The task of implementing fairness is simplified by the fact that finite execution paths are neither fair nor unfair. Finite paths are usually undesirable in a concurrent system and are reported as deadlocks. Only infinite paths are relevant to fairness and the only source of infinite paths is cycles in the state graph. Since every cycle forms an infinite path, *all* cycles must be detected and checked for fairness.

3.5.1 Impartiality

To implement impartiality, every cycle must be checked to contain at least one transition from each process (using the τ partition described in Section 2.1). One way of achieving this is by storing on the depth-first stack along with every state a counter for each process. A counter is incremented whenever the corresponding process executes a transition. When a cycle is detected the current set of counters is compared to the set of counters found on the stack. If one or more counters are equal the corresponding processes have not had an opportunity to execute and the transition that forms the cycle can be ignored, since it does not form part of an impartial path.

Impartiality is not very expensive to compute: one increment operation per transition and $O(p)$ comparisons per cycle are needed, where p is the number of processes. The counters are only needed for states on the stack; states that have been moved to the cache are known to satisfy the specification in an impartial way.

3.5.2 Strong fairness

Strong fairness is easily implemented by detecting strongly connected components (SCCs). An SCC is a subgraph of the state graph such that there is a path between any two states of the subgraph. In other words, every state is reachable from every other. Every state graph can be partitioned into a finite number of SCCs. For example, the state graph in Figure 2.5 consists of a single SCC, since every state can be reached from every other state. For every infinite path in a state graph, there is a single SCC that contains its infinite tail. (The states of its finite "prefix" that lie outside the SCC are not important in this case.)

The goal is therefore to detect each SCC and to check that the strongly fair paths it contains satisfy the correctness specification. If an SCC contains one infinite path that satisfies the correctness specification, all its strongly fair paths must satisfy it; the transition that leads to the state where the specification is true cannot be ignored indefinitely by any strongly fair path. To check for strong fairness, it is therefore sufficient to check that the correctness property is satisfied by at least one infinite path in each SCC. There are several algorithms for detecting SCCs; the most widely known is a linear-time algorithm due to Tarjan [50].

How is an SCC checked to contain an infinite, satisfying path? Associated with each state is a flag called *goodchildren*. This flag indicates that the correctness specification is satisfied by the marked state or one or more of its descendants. A state can propagate the flag up to its parent state, since, if it has any "good children", so does its parent. However, it cannot pass it down to its descendants. As soon as a state is found that satisfies the specification, its *goodchildren* flag is set.

Tarjan's algorithm ensures that an SCC is detected only when all its members have been explored and the root of the SCC is the current state. The model checking algorithm must postpone reporting errors until an SCC root has been found. If the *goodchildren* flag is set for the root state of the SCC, it implies that all strongly fair paths in the SCC satisfy the correctness specification. If the flag is not set, an SCC has been found in which the specification is not true for any path and therefore not for any strongly fair path. In this case, the validator will abort the analysis and report the error.

The detection of SCCs is not expensive in terms of runtime: Tarjan's algorithm operates in

$O(n)$ time, where n is the number of states in the state graph. Moreover, the algorithm can be modified to operate on-the-fly as the state graph is generated; such an implementation is described in the next chapter. However, this algorithm requires that states are retained on a stack until their entire SCC has been recognised. Depending on the structure of the state graph, this may cause a considerable increase in the memory needed to store the stack. In the worst case the entire state graph will be stored on the stack until the analysis is completed.

3.5.3 Weak fairness

It seems that built-in support for weak fairness requires significant computation. For every cycle that is detected the model checker must check that every ignored transition is not enabled in at least one other state of the cycle (therefore not continuously enabled), or that such a transition cannot lead to a satisfaction of the correct specification. Only then has a cycle been found that is weakly fair and violates the specification. Regrettably there is, as far as we know, no literature that addresses this question.

Chapter 4

Implementation of a model checker

This chapter describes the design and implementation of a practical model checker to illustrate the issues discussed in Chapter 3. The model checker is the third in a generation of model checkers [2, 13, 40, 58] developed at the University of Stellenbosch, and as such, there are several "givens", inherited from its predecessors:

- A structure-based on-the-fly model checking algorithm for a subset of CTL is used.
- Models are written is ESML (Extended State Machine Language) [14]. ESML was designed to meet the needs of reactive systems, namely complex data structures.

In addition, the following techniques were selected from those described in the previous chapter:

- The model checker uses abstract interpretation to execute transitions.
- State caching is used.
- States are represented using the state compaction technique described in Section 3.4.3.
- The model checker has built-in support for strong fairness.

Oberon was selected to implement the model checker [62]. Oberon is a strongly typed language that supports modularisation—two important features for developing any piece of large

software. The model checker comprises roughly 1900 lines of code and the ESML compiler a further 3700 lines.

The structure of the model checker is shown in Figure 4.1. As indicated, the model checking algorithm interfaces with the state generator (module `Machine`) only. The depth-first stack is implemented in module `Trace`, and the state cache in module `Cache`. All the modules of the state generator use module `Compact` which abstracts the `State` type.

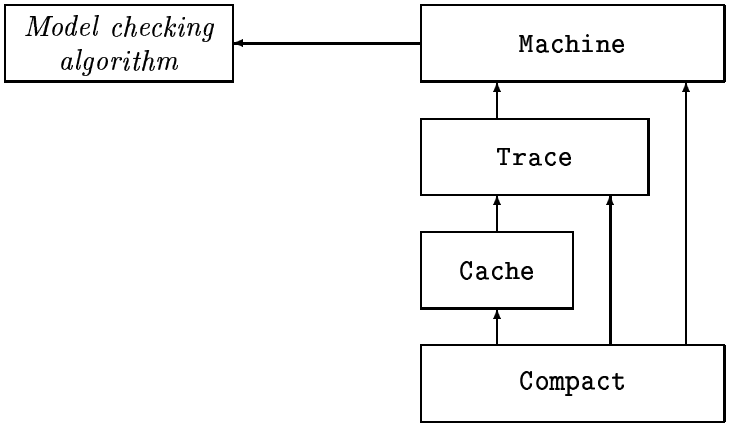


Figure 4.1: Module structure of the model checker

4.1 State generation

The state generator interacts with the model checker by means of three main routines: procedure `Execute` selects a process and transition and invokes the interpreter to execute the instructions. If a new state is generated, it is placed on the stack. `Backtrack` simply removes the top state of the stack. These operations correspond to moving down and up along an edge in the state graph. `Evaluate`(p) returns a Boolean result to indicate whether the atomic proposition p holds in the current state. Other, minor routines are used for secular duties such as reporting the number of states explored.

4.1.1 Procedures `Execute` and `Backtrack`

Procedure `Execute` is shown in Figure 4.2. Its actions are embedded inside a loop that iterates until it successfully generates a state, runs out of transitions, or detects a transition error.

```
1  PROCEDURE Execute(): INTEGER;
2  VAR result: INTEGER;
3  BEGIN
4     LOOP
5        Reschedule;
6        IF no (more) transitions enabled THEN
7           RETURN AllChildrenExplored or Complete
8        END;
9
10       result := Step(sch);
11       Trace.Update(sch);
12
13       IF result = Progress THEN
14          update the location counter of the executed process
15          CASE Trace.Push(state) OF
16             Trace.Inserted: RETURN Forward
17          |  Trace.Revisit: RETURN Revisit
18          |  Trace.Loop: RETURN Loop
19          END
20       ELSIF result = NoProgress THEN
21          (* do nothing *)
22       ELSIF result = TransitionError THEN
23          RETURN Error
24       END
25    END
26 END Execute;
```

Figure 4.2: Procedure `Execute`

Its first task is to find a transition to execute. It invokes procedure `Reschedule` (line 5): this routine searches for a process that is ready to execute a transition. If no executable transitions are found by `Reschedule`, procedure `Execute` will return `AllChildrenExplored` (line 7). This signals that the current execution path is a dead-end and the model checking algorithm is expected to call procedure `Backtrack` and explore other branches of the state graph. In the

special case where the entire state graph has been explored, `Complete` is returned in line 7.

When an enabled transition is found, it is executed by the interpreter, procedure `Step`, that is invoked in line 10. The `sch` variable is a record that stores scheduling information; it identifies the transition that was selected for execution. After `Step` has returned, the stack is updated by the call to `Trace.Update` to reflect the fact that the transition has been explored. It is critical to record this fact, so that, if the current state is reached again when falling back, the same transition is not re-explored.

The interpreter returns one of three values:

- `Progress` (line 13): The interpreter executed the transition and a new state has been reached. In line 14 the location counter of the executed process is updated and the state is pushed onto the stack in the next line. If this operation is successful `Execute` returns the value `Forward`. If the state is already present in the cache or on the stack, `Execute` returns `Revisit` or `Loop`, respectively. In this case the state is not added to the stack. The model checker may respond to the situation as it sees fit and then call `Execute` once again to generate further states.

- `NoProgress` (line 20): The interpreter was unable to complete the execution of the transition. This occurs when a communication instruction is identified by `Reschedule` as a potentially enabled transition, but upon further investigation by the interpreter it turns out not to be enabled. Control therefore returns to the top of the loop and another transition is investigated.

- `TransitionError` (line 22): The interpreter has encountered an error (such as division by zero, or the removal of the head element from an empty list) while executing the transition. In this case `Execute` returns the value `Error`, and the model checker aborts the analysis of the model.

In contrast to `Execute`, procedure `Backtrack` (Figure 4.3) is simple: it consists of one invocation of `Trace.Pop`. The call removes the top state of the stack and moves it into the cache of visited states. This implies that the state complies with the correctness specification. `Backtrack` is invoked by the model checker when it reaches a dead-end in the state graph, or when it has

explored an execution far enough to determine whether the correctness specification holds or not.

```
1  PROCEDURE Backtrack;
2  BEGIN
3     Trace.Pop(state, sch, store)
4  END Backtrack;
```

Figure 4.3: Procedure `Backtrack`

4.1.2 The architecture of the abstract machine

The design of the interpreter is based on the requirements of the ESML language, a complete description of which can be found in Appendix D. Special attention was paid to instructions to implement the language's support for lists as native data structures, concurrent processes, synchronous communication, and non-deterministic choice. The implementation of the abstract interpreter is straightforward: the data structures of the abstract machine, the implementation of procedure `Step` and the instruction set are discussed below. (The author wishes to thank Hans Loedolff who designed an initial version of the machine.)

Data structures

The key data structures of the abstract machine are shown in Figure 4.4.

The memory of the interpreter is called the `store`. The store holds the abstract code, variable space, and the expression stack. The abstract code starts at position 0 and contains the abstract machine code for the model. It is followed directly by the variable space, where all model variables and location counters are stored. The variable space is divided into regions called *frames*, with one frame allocated per process. The first word of each frame holds the process's *location counter* and the rest of the frame holds its local variables. The machine uses an expression stack that starts at the highest store address and grows down towards the start of the abstract code. The stack pointer `sp` holds the address of the top word of the stack.

The compacted state vector is stored in variable `state`. It contains an exact, compacted copy

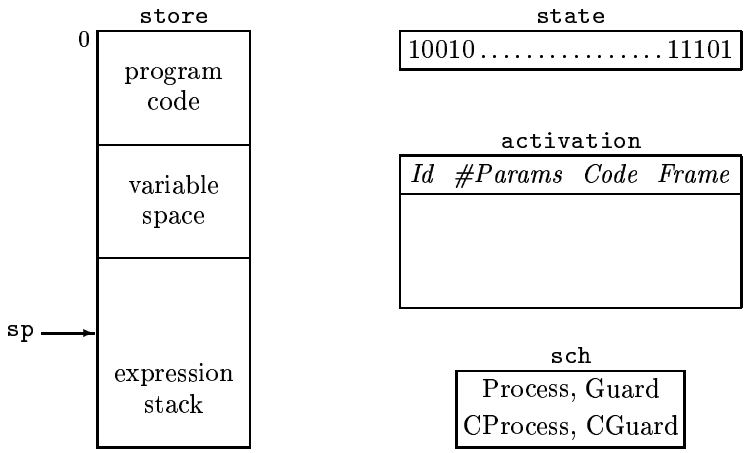


Figure 4.4: Key data structures of the interpreter

of the variable space, in other words, all local variables and location counters. Each program variable therefore has two representations in the abstract machine: it is stored once in the variable space and once in the compacted state vector. This duplication is necessary because the variable space is too large to act as the canonical representation of the current state—in the case of the mutual exclusion example of the previous chapter, the variable space is roughly ten times larger than the compacted state vector. On the other hand, extracting the values of variables from the state vector is too inefficient due to the large number of variable accesses and the fact that the state vector is compacted. To maintain the consistency of the state vector and the variable space, *both* the compacted state and the store must be updated whenever the value of a variable changes.

Information about active processes are store in an activation list called `activation`. For each process the activation list contains one entry that stores a unique number for the process, its number of parameters, code address and frame address. The process's location counter is not stored in this table, but in the first word of its variable frame. ESML does not allow dynamic creation of processes and the information in the table remains static during the analysis.

Lastly, the `sch` record stores scheduling information that determines which instruction the abstract machine is about to execute next. The record identifies the process that executed the last transition. The *Process* field stores the number of this process. In the case of `DO`,

`IF` and `POLL` constructs that contain several guarded commands it is necessary to identify the particular guard that was executed. This information is stored in the *Guard* field. Furthermore, a communication command can synchronise with more than one partner and more than one guard of a `POLL` command, and therefore the same information (*CProcess*, *CGuard*) is stored for the synchronisation partner. The state generator uses this record to select the next instruction to execute.

Procedure `Step`

The interpreter for the abstract machine is implemented by a single procedure called `Step`, which is outlined in Figure 4.5.

```
1  PROCEDURE Step(VAR sch: ScheduleInfo): INTEGER;
2  VAR transition: BOOLEAN; loc, result: INTEGER;
3  BEGIN
4     loc := store[activation[sch.process].frame];
5     transition := FALSE;
6     REPEAT
7        CASE store[loc] OF
8        |  instr_0: code_0
9        |  instr_1: code_1
10          ⋮
11       END
12    UNTIL transition;
13    RETURN result
14 END Step;
```

Figure 4.5: Procedure `Step`

The core of procedure `Step` is a fetch–decode–execute cycle. Line 4 calculates the address of the next instruction to execute (the fetch) and the case statement in line 7 decodes the instruction and selects the appropriate action (the execute). Each case interprets a single instruction: $\texttt{instr}_i$ is a constant that identifies the instruction and $\texttt{code}_i$ implements the meaning of the instruction. The `transition` flag indicates whether a transition has taken place, and the outcome of the transition execution is stored in `result`: this is either *Progress*, *NoProgress*, or *TransitionError*.

In general, each code fragment follows the same pattern:

1. It checks for transition errors (such as division by zero) and sets `result` if necessary;
2. it executes the action of the instruction (by making assignments to the various data structures of the machine); and
3. it advances the location counter `loc` and sets the `transition` flag if necessary.

The instruction set

The instruction set was designed to support efficient and reliable model checking, while meeting the needs of ESML. Since the machine is *abstract*, the instruction set is not constrained by typical considerations of hardware. Each instruction has only one format and the addressing modes are simple. This simplifies and speeds up instruction decoding. A stack-based instruction set was selected to simplify code generation as well as the implementation of instructions. The machine has no registers apart from `sp`, which can only be manipulated indirectly. When the value of an operand is fixed (for example, the target address for an unconditional jump), it is stored in the code directly after the instruction. Otherwise, an instruction fetches its operands from the stack where they are placed by the preceding instructions.

There are six classes of instructions.

1. *Arithmetic:* About one third of instructions implement arithmetic operations. These instructions are needed to evaluate ESML expressions.

   A typical instruction of this type is `add`, shown in Figure 4.6. The instruction replaces the top two elements of the expression stack by their sum. The diagram shows a snapshot of the store before and after the execution of the instruction. At the top of each snapshot the code and location counter is shown, and at the bottom the machine's expression stack and stack pointer.

   In principle, the add instruction should check that the stack contains at least two elements, but it is more efficient to rely on the ESML compiler to guarantee this condition. The instruction therefore performs no checks. Instead, it immediately adjusts the stack

```
1  | add:
2      INC(sp);
3      store[sp] := store[sp] + store[sp-1];
4      INC(loc)
```

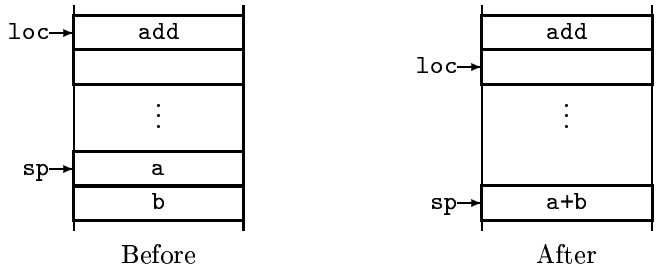


Figure 4.6: Implementation of the `add` instruction

pointer, performs the addition, and advances the location counter. It does not change either `result` or `transition`.

Other arithmetic instructions are the integer operations `sub`, `mul`, `div`, `mod` (remainder), `chs` (change sign); integer comparisons `compare`, `equal`, `neq`, `greater`, `geq`, `less`, `leq`; Boolean operations `and`, `or`, `not`; and expression subroutine instructions `evaluate`, `end`.

The instructions generated for expressions are not stored within the main body of code, but appears grouped in *expression subroutines* at the end of the abstract code. When the value of an expression is needed, an expression subroutine is invoked with the `evaluate` instruction. The subroutine returns to the caller when an `end` instruction is reached.

Figure 4.7 illustrates the use of expression subroutines. It shows the code generated for the assignment command `x := x - 1`. The address of variable `x` is 13. The `pushValue` instruction (address 105) places the target address for the assignment on the stack; `evaluate` invokes an expression subroutine to calculate the value of the right-hand side and place it on the stack; `popVariable` (address 007) removes two operands from the stack and stores the new value at variable address 13. When the subroutine is invoked, it places the value of `x` on the stack (addresses 100 and 102), places the constant 1 on the stack (address 103) and performs a subtraction (address 105). The `end` instruction at address 106 returns to the subroutine caller.

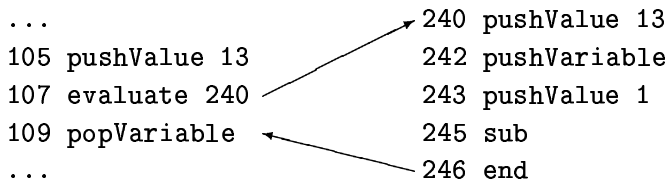


Figure 4.7: Code generated for the assignment `x := x - 1`

Apart from the `SKIP` command, all ESML commands (assignments, `IF`, `DO`, and communication commands) involve the evaluation of expressions. Expression subroutines were introduced to allow expression code to be translated to the native instruction set of the physical machine, but this idea has not been implemented (see Section 5.4.2).

2. *Memory transfer:* These instructions move values between the variable space and the expression stack. Variable addresses are relative to the frame of the current process and can be checked to ensure that a process does not read or write outside its frame. This check is not performed by the interpreter that instead relies on the compiler to perform scope checking. As an example of a memory transfer instruction, the implementation of the `popVariable` instruction, that removes a value from the stack and stores it in a variable, is shown in Figure 4.8.

   Line 2 calculates the variable address by adding the variable's offset `a` to the start of the current process's frame. The check in line 4 tests that the new value does not exceed the variable's range; `cardinalitymap` stores the maximum value of each location of the variable store. If this is not the case, the compacted state is updated (line 8) and the change is affected in the variable store (line 9).

   The other memory transfer instructions are `pushValue`, `pushVariable`, `pushVariableRange`, `popVariableRange`, `pushParameter`, and `pushParameterRange`.

3. *List manipulation:* The ESML languages incorporates lists as a primitive type. Initially the abstract machine handled lists orthogonally as just another data type that is manipulated on the stack. For example, the assignment

```
p := q :: <2> :: r
```

```
| popVariable:
    addr := activation[proc].frame + store[sp+1];
    (* addr now contains the position of the variable in the store *)
    IF (store[sp] < 0) OR (store[sp] >= cardinalitymap[addr]) THEN
       display message "Variable out of range"
       result := TransitionError
    ELSE
       State.UpdateValue(state, addr, store[addr] - store[sp]);
       store[addr] := store[sp];
       result := Progress
    END;
    INC(sp, 2);
    INC(loc);
    transition := TRUE
```

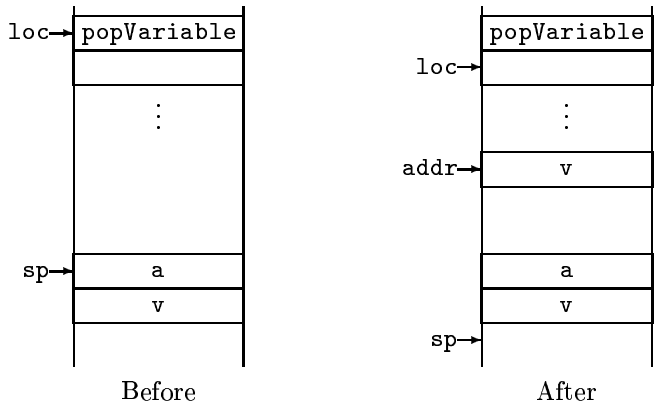


Figure 4.8: Implementation of the `popVariable` instruction

concatenates the lists `q`, `<2>`, and `r` and stores the result in list `p`. To execute this command the interpreter must push the contents of the three lists onto the stack, perform the concatenation operations twice and then move the result from the stack to the storage space for `p`. In addition to the list elements themselves, each list has a length attribute that must be manipulated correctly.

This approach proved complicated and inefficient. A study of existing ESML models revealed that the use of lists was restricted to the modelling of queues. The language was therefore modified to remove the general list operations of concatenation, list formation and list assignment. These were replaced by the following routines: `LEN(`*list*`)` returns the

length of the list, the `EMPTY(`*list*`)` routine empties the list, `HEAD(`*list*`)` returns the first element of the list, and the `REMOVE(`*list*`)` routine removes the head element. `APPEND(`*list*, *x*`)` and `PREPEND(`*list*, *x*`)` insert element *x* at either end of the list. These routines are implemented with the `length`, `head`, `empty`, `remove`, `append`, and `prepend` instructions. The `remove` and `prepend` instructions are still somewhat expensive, since they cause the contents of the list to be shifted one position forward or backward, but this cannot be avoided. The instructions are however much simpler than before.

4. *Control flow:* The ESML `IF` and `DO` control structures are translated with a `guards` instruction, followed by a code fragment for each of the construct's guarded commands. An example of code generated for an `IF` construct is shown in Figure 4.9. Each guarded command fragment starts with a (*guardExpr,nextGuard*) pair, such as those at addresses 102 and 107 in the example. The first number in the pair is the address of the expression subroutine that evaluates the guard; in the example, the subroutine at address h_1 evaluates g_1 and the subroutine at address h_2 evaluates g_2. The second number in the pair is the address of the next guard. Each pair is followed by the code for the corresponding action.

```
1  IF                          100 guards 112
2     g1 ->                    102 (h1, 107)
3           c1                 104 code for c1
4                              105 jump 113
5  [] g2 ->                    107 (h2, 112)
6           c2                 109 code for c2
7                              110 jump 113
8  END                         112 trap
9  ...                         113 ...
```

Figure 4.9: Code generated for an `IF` command

The code for an `IF` construct ends with a `trap` instruction that aborts analysis of the model when all guards are false, in keeping with the semantics of Dijkstra guarded commands. Of course, when an `IF` guard is satisfied and its action is executed, the analysis must not abort. Therefore, each action code c_i is followed by a jump over the trap. In the case of a `DO` construct there is no `trap` instruction; instead, each action ends with a

`jump` that directs the flow of control back to the start of the `DO`.

There are three other control flow instructions: `skip` encodes the ESML `SKIP` command, and `activate` and `terminate` create and destroy processes. Although the original definition of ESML allowed dynamic activation of processes, this feature has been eliminated to avoid the obstacles described in Section 3.2.4. Instead, the main body of the model (between the last `BEGIN` and `END` keywords) lists the processes that are active when the analysis begins, together with their parameters. This simplification has further benefits: ESML's original support for nested process definitions is rendered obsolete, since only globally visible processes can be activated by the main process, and it allowed the compiler to determine the number of processes and their order of activation during compilation, and to determine the allocation of bits in the state vector.

5. *Communication:* Send (`!`) and receive (`?`) commands are translated with `bang` and `hook` instructions. The `hook` instruction is interesting in that it is *passive* and never executes a transition. All communication is effected by the sending process. The `bang` instruction therefore scans through the list of active processes to search for communication partners, as described in Section 3.2.3.

   When a partner is found, the interpreter checks the other conditions before allowing the communication to succeed. Because of the semantics of communication in ESML, this process is quite complex. Consider the following two synchronising commands:

   $$channel\,!\,signal(data) \wedge condition1$$
   $$channel?signal(var) \wedge condition2$$

   After checking that the *channel* and *signal* values match, the interpreter evaluates *condition2*. If this is satisfied, the *data* expression is evaluated and the result is copied to the *var* variable in the receiving process's frame. The interpreter then evaluates the *condition2* expression. If this last condition is satisfied, the communication succeeds; otherwise, all changes are undone and the interpreter reports `NoProgress`. All expression evaluations involve the execution of expression subroutines.

   It is clear that these instructions are expensive to interpret, but this cost cannot be avoided. Simplifying the semantics of ESML was considered, but unfortunately the majority of models rely heavily on these commands.

The translation of the selective receive (`POLL`) command is similar to that of `IF` and `DO` discussed above. Instead of a `guards` instruction, a `poll` instruction is generated, and each guard pair contains a communication instruction. An example is shown in Figure 4.10. The expressions `y+1`, `x`, and `x>2` are evaluated by the subroutines at addresses h_{y+1} , h_x, $h_{x>2}$ respectively. The fourth operand of the `bang` and `hook` instructions is 1 in both cases and indicates the storage space required by the `y+1` and `x` expressions; the $\perp$ operand of the `bang` instruction means that there is no extra condition to be satisfied.

```
1  POLL                              240 poll 262
2     ch!a(y+1) ->                   242 (bang ch a h_{y+1}  1 ⊥, 252)
3           c1                       249 code for c1
4                                    250 jump 240
5  [] ch?b(x) & (x>2) ->             252 (hook ch b h_x 1 h_{x>2} , 262)
6           c2                       259 code for c2
7                                    260 jump 240
8  END                               262 ...
9  ...
```

Figure 4.10: Code generated for an `POLL` command

6. *Miscellaneous instructions:* The `outString`, `outValue`, `outRange` and `outLn` instructions provide a means of displaying values—an invaluable feature when developing models.

   Array manipulation is simplified by the `index` instruction that performs range checking and calculates the address of an array element. The implementation of `index` is shown in Figure 4.11.

   The instruction has two code operands: `s` is the array element size, and `m` is the array size. Array indices range from 0 to `m`−1. The array index and the array base address are placed on the stack. The test in line 2 checks that the array index `i` satisfies the array bound condition. If it is violated, the interpreter displays an appropriate message, and sets the values of `result` and `transition`. Otherwise, it adjusts the stack pointer, calculates the address of the element, and advances the location counter to the next instruction.

   The last special instruction is `selectProc`. When a process evaluates an expression, it may only access local variables and therefore all variables addresses are treated as offsets within the frame of the current process. In contrast, the CTL correctness specification

```
1  | index:
2        IF store[sp] >= store[loc+2] THEN
3           display message "Range check error"
4           result := Error;
5           transition := TRUE
6        ELSE
7           INC(sp);
8           store[sp] := store[sp] + store[sp-1] * store[loc+1];
9           INC(loc, 3)
10       END;
```

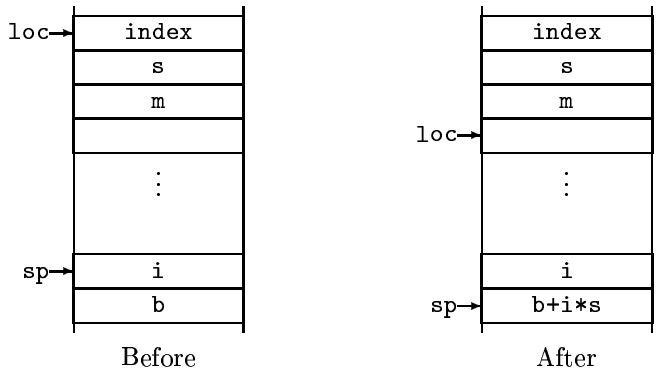


Figure 4.11: Implementation of the `index` instruction

can refer to any variable in any process. To select the appropriate frame in which a variable is located, the `selectProc` instruction is used.

Not all instructions can complete transitions and several instructions may execute before the `transition` flag is set. This amounts to a primitive form of atomic coarsening of actions [45] and is similar to the `d_step` construction used in the SPIN system [34]. The following conditions cause a transition to complete:

1. The value of a variable is changed
2. A communication instruction completes successfully
3. A guard evaluates to true
4. A `jump` instruction is executed

5. A transition error occurs or a `trap` instruction is executed

It is possible to coarsen transition even further: condition 1 can be made stronger by restricting it to variables that appear in the correctness specification, and condition 3 can be omitted.

4.1.3 The structure of the stack

The depth-first stack of the model checker is implemented in module `Trace`. During the analysis the stack stores current depth-first path and is used for cycle detection and for backtracking. Since it contains the exact sequence of states up to the current state, the stack can also provide the user with an error trail when a violation of the correctness specification is found.

When the model checker backtracks, the previous value of `state` is restored from the stack. It is also necessary to restore the machine's variable space to its previous value, but it is not viable to store the entire variable space on the stack because of its size. An alternative is to uncompact `state` and to reconstruct the variable in this way, but this introduces significant overhead. A third and simpler technique was adopted: all changes to variables are recorded and this record is then later used to undo all changes.

Every time a variable is changed, its address and previous value are pushed onto a special, independent stack called `delta`. Each entry of the depth-first stack maintains a pointer to the first of its changes. When the model checker backtracks, changes are popped from the `delta` stack and applied to the variable space until the first of the changes is reached.

The operation of this scheme is illustrated by Figure 4.12. In state s variable `x` is stored at address 13 and has the value 10. Transition t corresponds to the assignment `x := x-1`. When t is executed, address 13 and value 10 is stored in the `delta` stack and `x` is decremented. When falling back, the actions of transition t are undone: the last of these changes (there may be others) restores the value of `x` in the variable space to 10.

The `stack` data structure is implemented as an array of `StackEntry` records, shown in Figure 4.13. The `state` and `sch` fields store the state and scheduling information needed during the depth-first exploration; the `delta` field stores the address of the first of the changes that lead to the state stored in the stack entry.

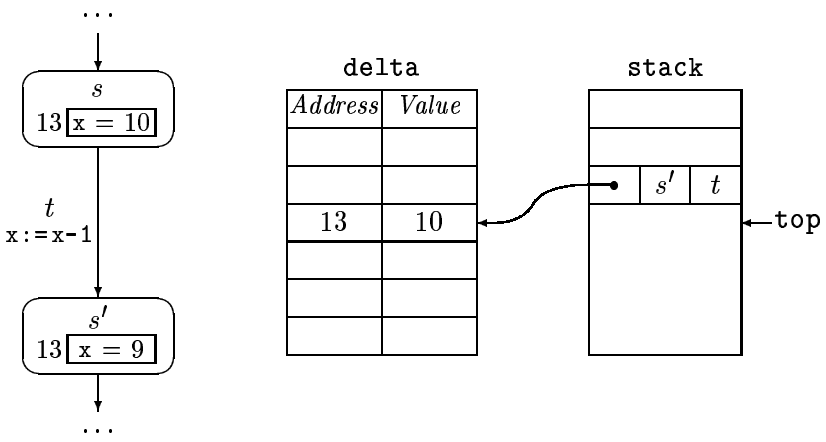


Figure 4.12: Illustration of the operation of the delta store

```
1  StackEntry = RECORD
2     state: Compact.State;
3     sch: ScheduleInfo;
4     delta: LONGINT
5  END
```

Figure 4.13: Definition of the `StackEntry` type

4.2 State caching

The model checker uses a state cache with hashing as described in Section 3.3.1.

An early version of the model checker stored stack and cache states separately. When a new state was not found in the cache, linear search was used to check whether the state appeared in the stack and formed a cycle. An execution profile revealed that the linear search accounted for the largest portion of model checking time, and prompted a new implementation of the stack. One option is to manage the stack as a second closed hash table similar to the state cache. The stack order is preserved by adding a field to each stack entry that threads the stack through the hash table. The main difficulty with this approach is that when a state has been fully explored, it must be removed from the hash table; this turn out to be an expensive operation.

An attractive alternative is store the stack states directly inside the cache. At first we were reluctant to tamper with the simplicity of the cache. Experience with an earlier model checker

had shown that adding flags to cache states can complicate the management of the cache beyond measure, but the inefficiency of linear search demanded action.

```
1  CacheEntry = RECORD
2     state: Compact.State;
3     flag: LONGINT
4  END
```

Figure 4.14: Definition of the `CacheEntry` type

Finally, the following scheme was adopted: Each entry in the cache consists of a state and a flag (Figure 4.14). If the entry is unused, the value of the flag is -2. Otherwise, the entry contains either a "stack state" or a true "cache state". For cache states, the flag is set to -1. A zero or positive value means that the state is part of the stack, and the value of the flag indicates the state's position in the stack.

When a new state is generated, it is searched for in the cache. If found, a negative flag signifies that the state is being revisited, while a positive values tells the position of the state in the stack. If a state is not found, it is immediately inserted with the stack pointer as its flag value. This task is performed by procedure `NewInsert`, which combines the functionality of the *Insert* and *Lookup* routines described in Section 3.3. Once a state has been fully explored and must move from the stack to the cache, its flag is merely set to -1.

The stack data structure is retained to store information about scheduling, backtracking and fairness. An additional field `pos` identifies the location of the state in the cache; the complete definition of stack entries is shown in Figure 4.15. The fields in lines 5 and 6 are discussed in Section 4.4.

```
1  StackEntry = RECORD
2     pos: LONGINT;
3     sch: ScheduleInfo;
4     delta: LONGINT;
5     pre, ll: LONGINT;
6     goodchildren: BOOLEAN
7  END
```

Figure 4.15: Modified definition of the `StackEntry` type

This implementation of the stack solves an important problem with regard to fairness. As mentioned in the previous chapter, a state must remain on the stack until its entire SCC has been detected. After the changes above these states are stored in the state cache where they need to be stored in any case if they satisfy the correctness specification. In the case where they violate the correctness property, they are never changed to "cache states"; rather, the analysis aborts as soon as model checking algorithm detects the violation.

Although the stack has only been modified in a minor way, its efficiency has improved dramatically. However, a new hurdle is introduced: "stack states" may not be overwritten by newer states, since this would corrupt the depth-first stack. This is easily avoided by testing that the flag is negative before overwriting a cache entry. The impact of this change on the efficiency of the cache is fully described in Section 5.2.2.

4.3 State compaction

The state compaction scheme described in Section 3.4.3 is used. As was noted there, an important attribute of this scheme is its simplicity and during the implementation only one complication arises: due to the large number of variables, it is not practical to store and manipulate the entire compacted state as a single number.

Variables are therefore grouped into *cells*. Each cell is compacted separately and variables may not be split over more than one cell. Variables are allocated to cells in their order of appearance and when the compacted number grows too large for a cell, the next cell is used. As a result, the last few bits of a cell may remain unused. The problem of finding an optimal arrangement of variables is equivalent to the bin packing problem and requires unaffordable overhead, though no new data structures would be needed to store such alternative variable orderings. In practice, the current strategy works well and few bits are wasted.

The `Compact` modules exports an abstract type `State` which is simply an array of words of type `LONGINT`. Each variable in the model is given a unique number, called its *index*. Two tables facilitate the implementation: `cellmap` maps each index to the word where it is stored, and `factormap` maps each index to its lower factor. Since the upper factor of variable i is equal to

the lower factor of variable $i+1$, it is unnecessary to store upper factors.

The implementations of the `GetValue` and `UpdateValue` operations are shown in Figures 4.16 and 4.17. Since the interpreter has access to the values of the variable store, `GetValue` is not used, except during initialisation. `UpdateValue` is almost identical to the *SetValue* routine described in Section 3.4.3: instead of the new value of the variable, it is invoked with the value $(v_i' - v_i)$, the difference between then new and the old value of the variable.

```
1  PROCEDURE GetValue(VAR s: State; index: INTEGER): INTEGER;
2  VAR k: LONGINT;
3  BEGIN
4     k := s[cellmap[index]];
5     IF factormap[index+1] > 1 THEN
6        RETURN SHORT(k MOD factormap[index+1] DIV factormap[index])
7     ELSE
8        RETURN SHORT(k DIV factormap[index])
9     END
10 END GetValue;
```

Figure 4.16: Procedure `GetValue`

```
1  PROCEDURE UpdateValue(VAR s: State; index, delta: INTEGER);
2  BEGIN
3     s[cellmap[index]] :=
4        s[cellmap[index]] + factormap[index] * delta
5  END UpdateValue;
```

Figure 4.17: Procedure `UpdateValue`

4.4 The implementation of fairness

Strong fairness has already been discussed in Sections 2.5 and 3.5. The latter explained the two elements of the implementation: the detection of strongly connected components (SCCs) using Tarjan's algorithm, and the use of the `goodchildren` flag to indicate whether or not an SCC satisfies the correctness specification. These elements are implemented by modifying the operation of the depth-first stack.

4.4.1 Simplifying assumptions

As noted, module `Trace` stores the current execution path in an array called `stack`. The stack pointer `top` points to the first open slot in the stack and the top element is `stack[top-1]`. For the sake of clarity, three simplifications about the structure of the stack are made:

The `delta` field is an index into a table which records all the changes made to the machine's variable store. When the model checker backtracks, this table is used to undo the changes and restore the variable store to a previous state. The operation of the `delta` field is straightforward, but since it is not relevant to fairness, it is ignored in the rest of this chapter.

The `goodchildren` field stores the flag that was described in Section 3.5. It plays a direct role in the fairness algorithm of the model checker, as the section explained. When a new state is pushed onto the stack, its `goodchildren` field is set to false; the flag of the top element can subsequently be tested with procedure `GetGoodChildren` and it can be set to true with procedure `SetGoodChildren`. When a state is popped from the stack and the flag is set, it is automatically propagated to the predecessor state by the stack. This flag is discussed at the end of this section, but will be ignored for the time being.

A last simplification concerns the integration of the stack and the cache that was discussed in Section 4.2. To simplify the presentation, the rest of this section ignores this optimization and assumes that stack entries have a field `state`. In the code that follows states are still explicitly inserted into the cache when they are popped from the stack.

4.4.2 Procedures `Push` and `Pop`

Stack entries are added and removed with procedures `Push` and `Pop` (shown in Figures 4.18 and 4.19).

`Push`: Procedure `Find` determines whether state `s` has been encountered before (line 4). It searches through the stack and then through the cache. If `s` is found in the stack, its position is assigned to local variable `k`, and `Loop` is returned (line 12). If `s` is not in the stack but in the cache, `k` is assigned -1, and `Revisit` is returned (line 10). The last possibility is that `s` is an entirely new state in which case `k` is assigned -2, the state is inserted into the stack, the

```
1  PROCEDURE Push(s: State): INTEGER;
2  VAR k: INTEGER;
3  BEGIN
4     k := Find(s);
5     IF k = -2 THEN        (* s is new *)
6        stack[top].state := s;
7        INC(top);
8        RETURN Inserted
9     ELSIF k = -1 THEN     (* s is revisited *)
10       RETURN Revisit
11    ELSE                  (* s is on the stack *)
12       RETURN Loop
13    END
14 END Push;
```

Figure 4.18: Procedure `Push`

stack pointer is advanced, and `Inserted` is returned (lines 6–8).

```
1  PROCEDURE Pop();
2  BEGIN
3     DEC(top);
4     Cache.Insert(stack[top].state)
5  END Pop;
```

Figure 4.19: Procedure `Pop`

`Pop`: As soon as a state is popped from the stack, it is moved to the cache. "Invalid" states are never inserted in the cache: when a violation of the correctness specification is found, the model checking algorithm does not pop the state, but aborts the analysis immediately and dumps the states of the current execution path (`stack[0..top-1]`) to a file instead. From this information the user can reconstruct the events that led to the error.

4.4.3 Tarjan's original algorithm

Tarjan's algorithm [1, 50] relies on depth-first numbering to find strongly connected components (SCCs). It is a recursive algorithm that explores a directed graph in depth-first order and

numbers the vertices as they are visited. Figure 4.20(a) shows an example of such numbering and illustrates the four types of edges that are found in state graphs:

1. *Tree edges* lead to new vertices found during the search (represented by the solid lines in the graph);
2. *Forward edges* lead from ancestors to descendants but are not tree edges (represented by the dotted lines from vertex 1 to 4 and from vertex 5 to 4);
3. *Back edges* lead from descendants to ancestors, possibly from a vertex to itself (the dashed line from vertex 3 to 1); and
4. *Cross edges* connect vertices that are neither ancestors nor descendants of one another (the dashed line from vertex 4 to 3).

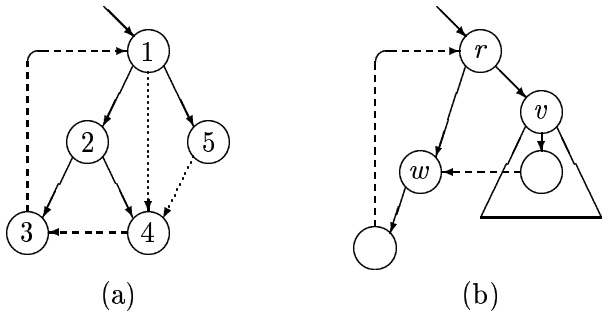


Figure 4.20: Depth-first search of state graphs

The first step in the development of the algorithm, is the insight that each SCC forms a subtree of the depth-first tree. Therefore each SCC has a "root," its smallest (shallowest) vertex. The algorithm uses depth-first numbers to find these roots and so to identify the SCCs. To aid in finding the roots a function called *Lowlink* is defined as follows ($\overline{v}$ is the depth-first number of vertex v):

$$Lowlink(v) = \min(\{\overline{v}\} \cup \{\overline{w} \mid \text{there is a cross edge or back edge from a descendant of } v \text{ to } w, \text{ and the root of the SCC containing } w \text{ is an ancestor of } v\})$$

Figure 4.20(b) illustrates the condition in the last part of the function. A cross edge leads from a descendant of v to w, where the root r of the SCC containing w, is an ancestor of v.

In [1] it is proved that a vertex r is the root of an SCC if and only if $Lowlink(r) = \overline{r}$. Procedure `Tarjan` in Figure 4.21 implements a recursive depth-first search that calculates *Lowlink* and consequently also all the SCCs.

```
1  PROCEDURE Tarjan(v: Vertex);
2  VAR w, x: Vertex;
3  BEGIN
4     INC(nodes);
5     v.dfnr := nodes;
6     v.lowlink := v.dfnr;
7     v.marked := TRUE;
8     push v on stack;
9     FOR each descendant w of v DO
10       IF NOT w.marked THEN
11          Tarjan(w);
12          v.lowlink := MIN(v.lowlink, w.lowlink)
13       ELSIF (w.dfnr < v.dfnr) AND w is on the stack THEN
14          v.lowlink := MIN(v.lowlink, w.dfnr)
15       END
16    END;
17    IF v.lowlink = v.dfnr THEN
18       REPEAT pop x from stack UNTIL x = v
19    END
20 END Tarjan;
```

Figure 4.21: Tarjan's original algorithm

The code consists of three parts: lines 4–8 initialise the attributes of a new vertex, lines 9–16 contain code to explore descendants, and lines 17–19 contain code that is executed after a vertex has been fully explored. Variable `nodes` counts the number of vertices encountered so far. The value of the `lowlink` attribute is calculated in three places. In line 6 the value is initialised to the depth-first number of `v`. In line 12 the (possibly smaller) *Lowlink* value is propagated back to a predecessor after a descendant has been explored, and line 14 calculates the value for a cross or back edge (the scenario in Figure 4.20(b)).

Lemmas in [1] prove that when an SCC is found, its vertices occupy the top entries of the stack and can all be popped at once. The root is the last vertex removed from the stack. In line 18 the states of the SCC are repeatedly popped until the root is reached.

4.4.4 Integration into `Push` and `Pop`

The algorithm in Figure 4.21 can be merged with procedures `Push` and `Pop` after making the following observations:

1. Line 11 is unnecessary, since the recursive calls to procedure `Tarjan` have been replaced by calls made to `Push` and `Pop` from outside the module.

2. The original algorithm uses two stacks: an explicit stack is manipulated in lines 8, 13 and 18, and an implicit procedural stack is used by the recursive call in line 11. The current depth-first stack called `stack` corresponds to the procedural stack. The explicit stack is used by the original algorithm to store the elements of an SCC until the entire SCC has been detected.

   It is unnecessary to create an additional stack for SCCs; the behaviour of the existing depth-first stack can be modified to perform this task. One physical stack is used to implement two conceptual stacks, the depth-first stack and the SCC stack. All new states are added to the physical stack and therefore to both conceptual stacks at once. States are removed from the depth-first stack once they have been explored, and are later removed from the SCC stack when the entire SCC has been explored. States remain longer on the SCC stack. In other words, the depth-first stack contains a subset of the states on the SCC stack. The SCC stack retains the normal ordering meaning that the predecessor of `stack[n]` is `stack[n-1]`. The stack pointer `top` points to the first empty slot just beyond top SCC stack element. For the ordering of the depth-first stack a `pre` field is introduced to point to a state's depth-first predecessor. The `pre` field defines the thread of the depth-first stack that winds through the states of the SCC stack. A new variable `dftop` points to the top element of the depth-first stack.

   This scheme is illustrated in Figure 4.22: there are seven states numbered 0 to 6 on the physical stack, all of which belong to the SCC stack, and `top` points to the first empty

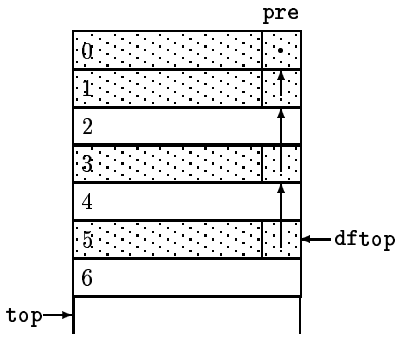


Figure 4.22: SCC and depth-first stack

slot. Only the gray states (0, 1, 3 and 5) belong to the depth-first stack. The `pre` field indicates the predecessor of each state and `dftop` identifies the top element of this stack. If state 5 is popped from the depth-first stack, `dftop` is updated to point to state 3. State 5 however remains on the stack as part of the SCC stack.

3. The value of `lowlink` for a particular vertex is not calculated by a single expression, but it is assigned an initial value in line 6 and is adjusted in lines 12 and 14 as information becomes available. The *Lowlink* value for a particular state can change over time and cannot be stored as a local variable in either `Push` or `Pop`. Therefore each element of the stack must store its own copy of its *Lowlink* value. A `lowlink` field is added to the definition of each stack entry.

4. The original algorithm uses the `marked` flag to determine whether a node is new or being revisited. In procedure `Push` this information is provided by the `Find` procedure. It is therefore unnecessary to include an explicit flag in the new algorithm.

5. The original algorithm uses depth-first numbering to (1) detect back/cross edges and (2) to provide an ordering of the vertices. The first task can be accomplished by searching through the stack—if a state is already present on the stack, it represents a back or cross edge. The second task is performed by ordering vertices according to their position in the physical stack. This ordering is only valid while a vertex remains on the stack, but this is also the only time during which the ordering is needed. The position in the stack indicates the order in which vertices were encountered, and therefore the stack ordering

is equivalent to depth-first ordering in this respect.

The modified algorithms

Procedure `Tarjan` in Figure 4.21 can now be inserted piecewise into the new versions of `Push` and `Pop` shown in Figures 4.23 and 4.24. Unless noted otherwise, line numbers refer to the algorithm in Figure 4.21.

Lines 4–5 and 7 can be omitted since depth-first numbers are not calculated explicitly and vertices are not labeled with the `marked` flag anymore. The actions of line 6 and 8 are performed by procedure `Push` in the case of a new vertex.

The `FOR`-loop in line 9 specifies that the code in lines 10–15 must be executed *for each descendant of v*. Both procedures `Push` and `Pop` are called once for each state (or descendant), so it is clear that the code must be inserted into these procedures. The first test in line 10 checks for new states and therefore seems to belong with lines 6–8 of Figure 4.18. Closer inspection, however, shows that the resulting action (line 12) happens after the recursive call to `Tarjan` has completed (after it has been explored), when vertex `w` propagates its value to its parent, vertex `v`. It therefore belongs in procedure `Pop`. The second `IF` in line 13 applies its action to back or cross edges and therefore belongs in procedure `Push` just before `Loop` is returned in line 12 of Figure 4.18.

Finally, lines 17–19 are performed after vertex `v` has been fully explored and they therefore belong at the end of procedure `Pop`. The test in line 17 to check whether an SCC has been found, is replaced by `stack[dftop].lowlink = dftop`. The removal of the top element in lines 3–4 of the original `Pop` procedure (Figure 4.19) is guarded by this test to mirror the conditional removal of states as in Figure 4.21.

This implementation of Tarjan's algorithm changes the runtime efficiency of the stack operations by only a constant factor. Memory is of course affected more seriously, since in some cases many states are retained on the stack. This issue is addressed in the next chapter.

```
1  PROCEDURE Push(s: State): INTEGER;
2  VAR k: INTEGER;
3  BEGIN
4     k := Find(s);
5     IF k = -2 THEN        (* s is new ≡ tree edge *)
6        stack[top].state := s;
7        stack[top].lowlink := top;
8        stack[top].pre := dftop;
9        dftop := top;
10       INC(top);
11       RETURN Inserted
12    ELSIF k = -1 THEN     (* s is revisited ≡ forward edge *)
13       RETURN Revisit
14    ELSE                  (* s is on the stack ≡ cross/back edge *)
15       stack[dftop].lowlink := MIN(stack[dftop].lowlink, k);
16       RETURN Loop
17    END
18 END Push;
```

Figure 4.23: Modified procedure `Push`

```
1  PROCEDURE Pop();
2  VAR pre: INTEGER;
3  BEGIN
4     pre := stack[dftop].pre;
5     stack[pre].lowlink := MIN(stack[pre].lowlink, stack[dftop].lowlink);
6     IF stack[dftop].lowlink = dftop THEN
7        WHILE top > dftop DO
8           DEC(top);
9           Cache.Insert(stack[top].state)
10       END
11    END;
12    IF (Good >= dftop) THEN Good := pre END;
13    dftop := pre
14 END Pop;
```

Figure 4.24: Modified procedure `Pop`

The `goodchildren` flag

One line of procedure `Pop` in Figure 4.24 remains unexplained. When the model checking algorithm determines that the correctness property is satisfied by the current state, it sets the `goodchildren` flag by calling procedure `SetGoodChildren`. Until now it was intimated that an individual flag is stored on the stack for each state. However, it is sufficient to store a single pointer to indicate the deepest stack position with this property.

When a new state is added to the stack, the *goodchildren* pointer `Good` remains at its current position, since it is not known whether the new state satisfies the specification. `SetGoodChildren` simply sets the pointer to the current top stack state. When the top state is removed from the stack and the state has the *goodchildren* property, `Good` is set to its predecessor, as is done in line 12 of Figure 4.24.

Chapter 5

Evaluation

Until now, design decisions have been motivated with arguments about conceptual clarity and efficiency. This chapter investigates whether these arguments hold true. It addresses the following questions:

- Is state compaction affordable? Do memory gains outweigh runtime overhead, or should compaction be removed?
- Does the performance of the cache/stack design correspond to results reported in the literature such as [26]?
- What is the runtime overhead incurred by strong fairness?
- What is the runtime overhead incurred by interpretation?
- How does the composition of the instruction set influence the performance? Which instructions are inefficient and how can the abstract machine be improved?
- How does the model checker compare to a more established system like SPIN?

Results are summarised at the end of the chapter.

Experiments were conducted on an SGI Indy workstation with a 150MHz MIPS R4400 processor and 64 megabytes of physical memory. Swapping is avoided by minimising the system load to

guarantee access to as much physical memory as possible. Time measurements are made using a standard Unix system call, `getrusage`, that returns information about resource usage. All measurements include the time spent inside the model checker process and the time spent by the operating system executing the model checker's system calls, but it excludes time waiting for IO request completion. In all cases the elapsed physical time is close to the given times (within 4% of). All time measurements are given in seconds and are averaged over a number of runs.

Care was taken to select representative ESML models. When modelling a reactive system, it is sound practice to start out with a restricted model and extend it in several refinement steps. It is therefore important to know how the model checker behaves when the scale of a model is increased. Four models were selected to make measurements, they appear in Table 5.1 and their source code can be found in Appendix A. They were chosen to represent different levels of communication activity and data requirements. For example, the dining philosophers models contain little data and a medium amount of communication, while the process scheduler models contain a high amount of both data and communication. The first three model types are parameterised in the number of processes, and the sliding window protocol models in the size of the window. In total there are 12 variations that range in size from approximately 8500 to 2.6 million unique states.

Model	*Description*
DPn	Classical problem of n dining philosophers, $n = 7, 8, 9$
ELn	Elevator model for n floors, $n = 3, 4$
PSn	Process scheduler for n processes, $n = 1, 2, 3$
SWn	Sliding window protocol for n-slot window [54], $n = 1, 2, 3, 9$

Table 5.1: Models selected for performance measurements

Except where noted otherwise, models are analysed for deadlock freedom, and runs are *perfect*, meaning that the state cache is made large enough to ensure that no states are ever replaced. Imperfect runs are investigated in Section 5.2. In most of the sections that follow, results are given only for selected models that exhibit typical behaviour; full results can be found in Appendix B.

5.1 State compaction

State vectors are implemented as an abstract data type by the `Compact` module. As the name implies, the state operations use the technique for compacting states as described in Section 3.4.3. However, since all these operations are fully encapsulated by this module, their implementation can be changed without affecting the rest of the system. This allows alternative forms of state compaction to be tested. In this section, the state compaction technique is compared to an implementation that performs no compaction at all, to determine what the cost of performing compaction is.

5.1.1 Number of compaction operations

Table 5.2 presents a count of how many state operations were executed. It contains information about the sliding window protocol models (SWn), their number of states and transitions and the number of `Compare`, `Assign` and `UpdateValue` operations. `Compare` tests whether two state vectors are equal, `Assign` assigns one compacted state vector to another, and `UpdateValue` updates the value of one of the variables stored in a state vector.

Model	*States*	*Transitions*	`Compare`	`Assign`	`UpdateValue`
SW1	22464	98449	90373	22464	121998
SW2	100426	447608	539143	100426	563375
SW3	235098	1052442	1679839	235098	1341667
SW9	2673976	12050941	25313655	2673976	16382794

Table 5.2: Count of compaction operations for the sliding window protocol models (SWn)

The most apparent feature of this table is the exact correspondence between the number of unique states and the number of `Assign` operations. Since these runs are all perfect, each state is inserted into the cache exactly once, and this is the only time that the `Assign` operation is used.

A second observation is that the number of `UpdateValue` operations is of the same order as the number of transitions. The `UpdateValue` operation is used to update the location counter, which happens once for every transition. It is also used for variable assignments and from

this fact the number of variable assignments can be calculated. For example, SW3 produces $1341667 - 1052442 = 2898225$ variable assignments.

Lastly, the absence of `GetValue` operations, that were described in Section 3.4.3, is noted. When the value of a variable is needed, it is not extracted from the compacted state with a `GetValue` operation, but taken directly from the variable space instead. Since the state vector and the variable space are kept consistent, the value is the same. The time spent on the operations is discussed in the next section.

5.1.2 Validating without compaction

It seems reasonable to assume that there is a fair amount of overhead involved with compaction: removing the redundancy from the uncompacted state vector requires processing. Moreover, the goal of compaction is to improve memory usage, even at the (very likely) expense of processing time. It is therefore expected that the speed of the model checker will increase when compaction is removed.

Very few changes are needed to disable compaction. `Compare` and `Assign` perform no calculations on the state vector, but merely compare and copy it word for word, and their execution times are directly proportional to the size of the state vector. In fact, the code for `Compare` and `Assign` remains unchanged. The `UpdateValue` operation modifies only a single word of the state vector and therefore its execution time is constant. When compaction is disabled, the `UpdateValue` operation does not make use of the formulas of Section 3.4.3 anymore, but stores the values directly in the state vector with a simple assignment.

To measure such a speed increase, these changes were made; the results of this experiment are shown in Table 5.3. Since the changes are transparent to the rest of the model checker, the usage counts in Table 5.2 remain exactly the same. The largest models (EL4 and SW9) could no longer be run, due to their excessive memory requirements.

Surprisingly, the speed of the analysis *decreased*: the models took between 1.25 (DP8) and 1.84 (EL3) times *longer* to analyse without compaction, and required more than three times more memory. The explanation for the decrease in runtime lies in the fact that the longer state

	Compaction		*No compaction*	
Model	*Bits*	*Time*	*Bits*	*Time*
DP7	50	8.11	272	10.19
DP8	54	29.35	304	36.83
DP9	58	102.20	336	133.52
EL3	89	10.76	432	19.78
EL4	107	341.18	·	·
PS1	59	1.13	320	1.70
SW1	43	3.15	240	4.71
SW2	53	13.79	288	20.38
SW3	56	33.74	336	34.35
SW9	89	471.89	·	·

Table 5.3: Performance of the model checker with and without compaction
(Time measurements are given in seconds)

vectors are costly to compare and assign. The time saved by not compacting is negated by the overhead of `Compare` and `Assign` operations. The same behaviour was observed when the author modified the SPIN system to use this state compaction technique [22].

The size of the uncompacted state is equal to the size of the variable space, and in fact the state vector and the variable space are identical. Every component of the state vector is stored in one 16-bit word, and therefore the uncompacted state size is a multiple of 16. It would be possible to store variables more compactly according to one of the methods mentioned in Section 3.4, but only at the expense of increased running time.

5.2 State caching

Apart from the complexity of the model checking algorithm, the most important factors in the performance of the model checker are the storage of states in the cache and the generation of states. In this section, the parameters of cache performance are evaluated.

5.2.1 The influence of cache size

How do we expect the cache with its closed hashing scheme to behave? (1) If the size of the cache is larger than the size of the state space, the cache acts as a perfect store. Collisions are exceptional and hardly any states should ever be replaced. (2) If the cache size is slightly less than the size of the state space, collisions are more frequent and a small number of states are replaced, but cache performance is still acceptable. (3) If the cache size approaches a certain critical size (which depends on the particular model, but is usually about half of the state space size), the number of collisions and replacements increases dramatically. (4) A cache size less than the critical size leads to an explosion in the number of transitions caused by the cache only storing a small region of the state graph. This forces the model checker to unnecessarily re-analyse many if not most states.

To confirm this description, the EL3 model was analysed 10 times with different cache sizes. The results are shown by the solid curve in Figure 5.1. The cache size started at 90000 states and was decreased in each run by a variable amount. The number of transitions explored is used as a measure of performance. This provides more accurate results and contains more information that time measurements, but in all cases the running time follows a similar pattern.

While the cache size is greater than the number of unique states, cache performance is satisfactory. As the cache size approaches the number of states however, performance deteriorates and even for a cache size only slightly less than the number of states, the number of transitions is unacceptably high. The $\triangle$-symbol at the top of the solid curve marks a point at which the cache size is 99.3% of the state space, but the number of transitions is about 2.5 times the normal figure of 323839.

The outcome of this test was in sharp contrast with the description of expected behaviour and with the results reported in [26]. The sudden, dramatic increase in the number of transitions means that many states are revisited, but not found in the cache, and are therefore re-explored, leading to the escalation in the number of transitions. A possible explanation for the discrepancy is a higher number of visits per state. In [26] the authors claim that each state is visited 3 times on average, based on the ratio of transitions to unique states. The corresponding ratio for the 12 representative ESML models ranges from 3.8 to 5.6, and for the EL3 model it is

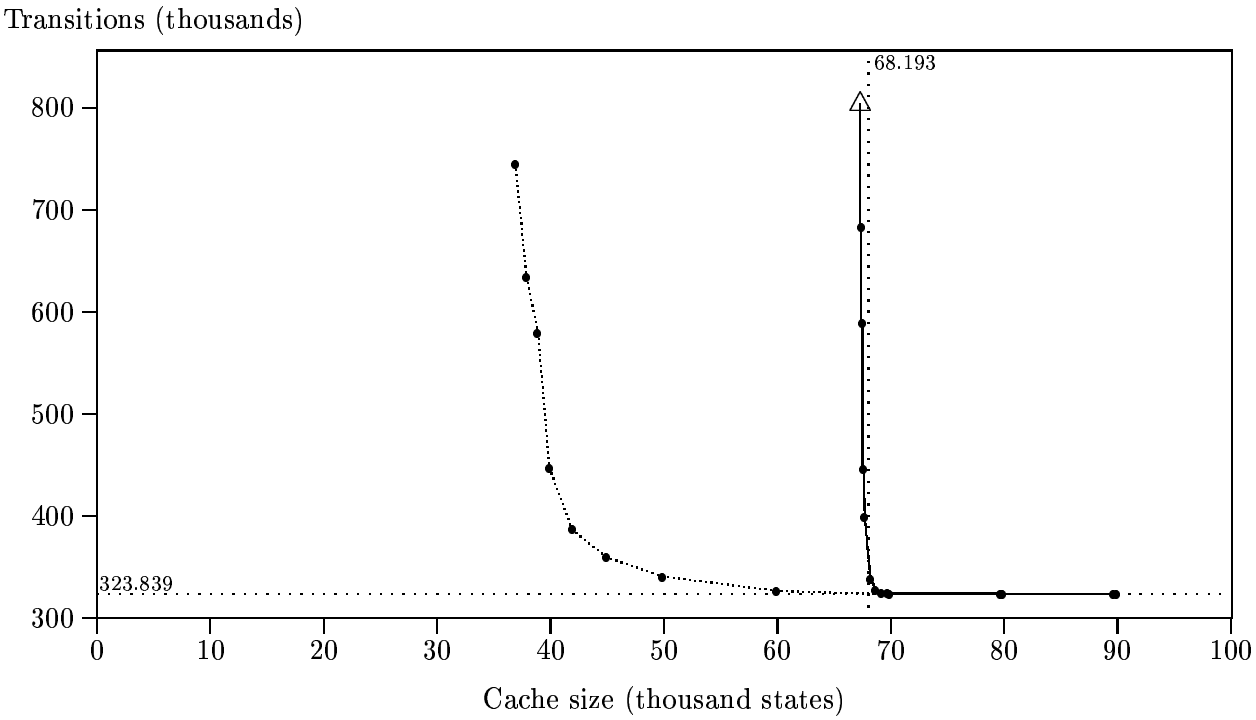


Figure 5.1: Influence of cache size on performance for the elevator model (EL3)

4.75, which is certainly higher than the cited value of 3. In fact, it is possible to measure the distribution of state visits, as shown in Figure 5.2. The histogram reveals that very few states are visited only once, and that there are even states with up to 14 visits. Close to three quarters of the states are visited four or more times.

How could this hypothesis—that the different behaviour of the cache is due to the high ratio of transitions to unique states—be tested? It is not possible to manipulate the average number of revisits in a particular model: this is an inherent property of the state graph produced by the transition semantics of ESML. The user has no direct control over the state space. The investigation of this problem led to the discovery of a programming error in the `Cache` module. The erroneous code disregarded all replacements: instead of overwriting the older state, both the older and the newer state were discarded.

This example of debugging underscores the importance of thorough measurements of a model checker: intricate errors are not always visible or are easily misinterpreted during the normal operation of the system.

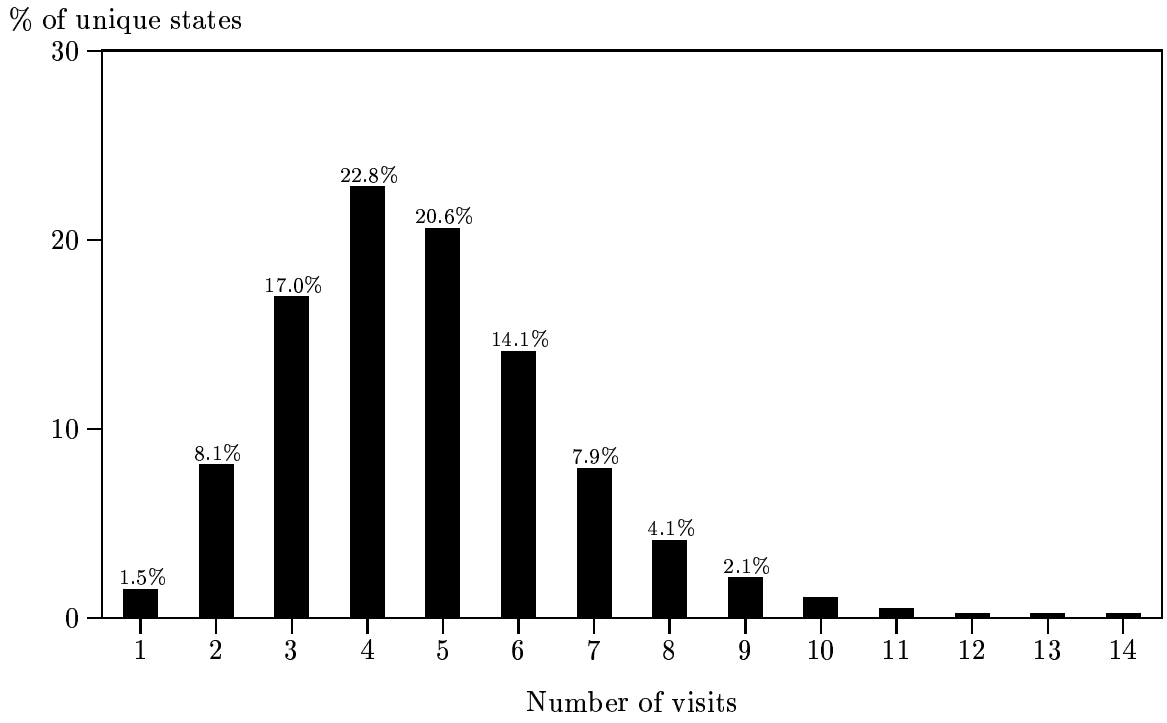


Figure 5.2: Number of visits per state for the elevator model (EL3)

After the error was corrected, the test was repeated, and the corrected results are shown by the dotted curve in Figure 5.1. The corrected cache now exhibits the behaviour reported in [26]: the cache remains effective for cache sizes less than the number of unique states, until the cache size approaches a critical size (roughly 60% of the state space size), where the performance deteriorates rapidly. Before 50% is reached, the cache becomes totally ineffective. In general, the cache can handle state spaces 1.7 to 2.0 times the size of the cache. The figure is slightly lower than that reported in [26], but this discrepancy can be explained by the state revisit ratio discussed above.

The curve in Figure 5.1 may create the impression that the number of transitions increase smoothly as the size of the cache is decreased. However, a more accurate picture is shown in Figure 5.3. The presentation is slightly different: along the horizontal axis the cache size is shown as a ratio to the number of unique states (*cache size*/68193), while along the vertical the number of transitions is shown as an inverse ratio to the number of transitions in the state graph (323839/*transitions*).

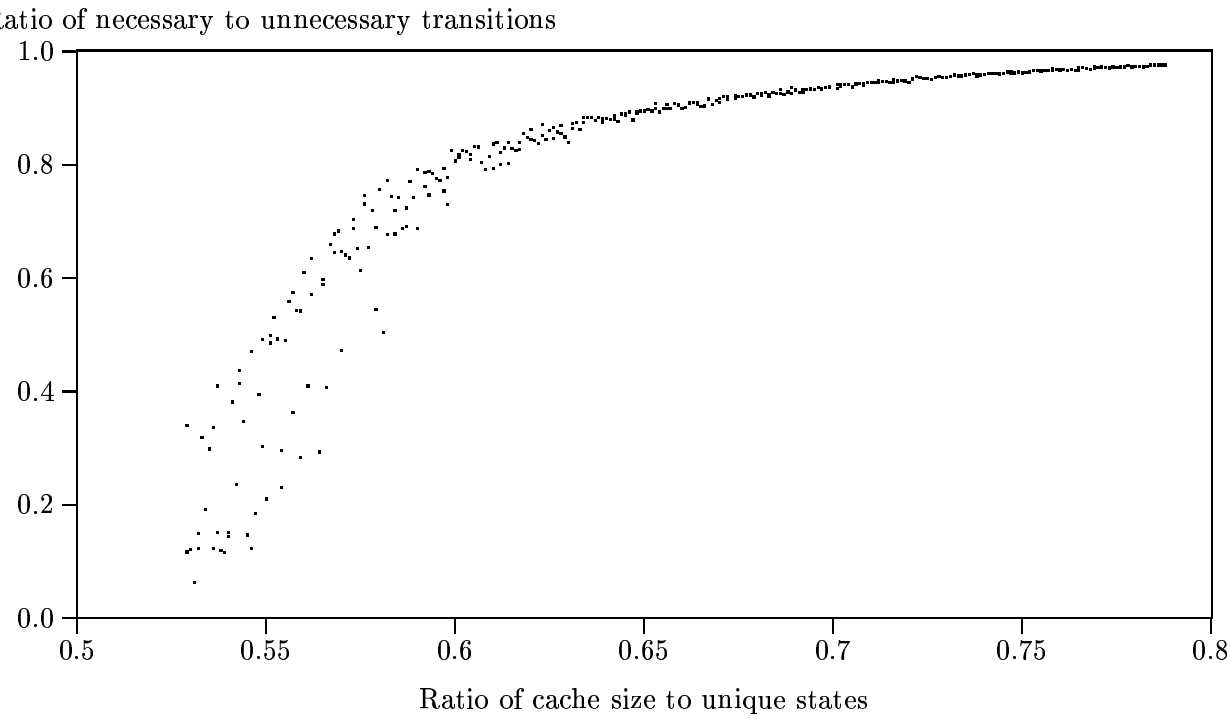


Figure 5.3: Detailed measurement of influence of cache size on transitions for the elevator model (EL3)

The small fluctuations in Figure 5.3 are due to the influence of the cache size on the hash function. Since the hash value of a state is calculated modulo the cache size, different cache sizes affect the distribution of the hash function. This causes different states to be replaced in each run. In some cases replaced states are visited again, and in other cases they are not; the cumulative effect of these differences explains the variations observed.

5.2.2 The influence of tolerance parameters

Size is not the only factor in the performance of the state cache. This section considers the cache insertion algorithm shown in Figure 5.4. It takes as input a state `s` and returns either *revisit* or *new*, or aborts the analysis with a message that the cache is full. Here *revisit* means that the state is either a regular cache state or a stack state, in which case a cycle has been detected.

Variable `x` stores the primary hash value for state `s`, and `d` the secondary hash value which will

```
1  PROCEDURE NewInsert(s: State);
2  VAR x, d, n: LONGINT;
3  BEGIN
4     x := hash1(s);
5     d := hash2(s);
6     n := 0;
7     WHILE n < C DO
8        IF cache[x] = s THEN RETURN revisit END;
9        IF cache[x] is empty THEN cache[x] := s; RETURN new END;
10       x := (x + d) MOD size;
11       INC(n)
12    END;
13    (* ready to replace another state if necessary *)
14    WHILE n < P DO
15       IF cache[x] = s THEN RETURN revisit END;
16       IF cache[x] is replaceable THEN cache[x] := s; RETURN new END;
17       x := (x + d) MOD size;
18       INC(n)
19    END;
20    Abort the analysis---the cache is full
21 END NewInsert;
```

Figure 5.4: Insertion algorithm for the state cache

be added to `x`, if necessary; variable `n` counts the number of collisions. The first while-loop resolves collisions until a certain number of them has been encountered, and the second while-loop searches for a replaceable state. In line 16 *replaceable* means that the cache slot is empty or contains a regular cache state. Therefore, the assignment in line 16 is not necessarily a replacement—the cache slot could be empty. (As an aside, the error referred to in the previous section, was caused by switching lines 15 and 16, thereby masking out *revisit*s.)

The algorithm contains two parameters that determine how "tolerant" the cache behaves: C (line 7) specifies the number of collisions that is allowed before the algorithm is prepared to let the new state replace another, older state; and P (line 14) is the total number of unsuccessful probes that is allowed before the algorithm concludes that the cache is full and aborts the analysis. It is clear that $0 < C \leq P$.

When the value of C is small, the algorithm is quick to decide that `s` is a new state and only a

few collisions are tolerated before `s` replaces another state. This behaviour leads to clustering around the collision slots. Although each invocation of the algorithm is fast, many more states are replaced and the algorithm has to be invoked many more times. On the other hand, when C is large and the cache fills up, many collisions may be resolved to find an empty slot. It takes a long time to insert each state, but the algorithm makes thorough use of the cache and fewer replacements are performed.

If the stack states were not stored in the cache, the P parameter would be unnecessary, since the test in line 16 would be true before the first iteration of the while-loop. As long as the number of stack states is small relative to the size of the cache, the iterations of the second while loop stay small—it never has to seek long to find a non-stack slot.

When $P = C$, no replacements whatsoever are allowed. In this case, the larger the value of C (and P) the more use is made of the cache; maximal use is made of the cache, when $C = \textit{cache size}$. When P is only slightly larger than C, the algorithm is quick to abort the analysis, often unnecessarily, since a few more probes might find an empty slot. When P is much larger than C, the algorithm again makes very thorough use of the cache, but the extra work performed may be unnecessary since, with little room in the cache, it is likely that the analysis is aborted anyway as the exploration of the state graph progresses.

For the analyses in this chapter the value of C is 100 and P is 200. However, these values are not relevant, since in the perfect runs, the value of C is never exceeded, and in the imperfect runs in this section P is never exceeded.

5.3 The cost of fairness

The model checker supports strong fairness with two mechanisms: Tarjan's algorithm for detecting SCCs and the `goodchildren` flag for recording the presence of satisfying paths. The authors of [25] are pessimistic about the performance of this approach; instead, they adopt an alternative that requires less memory, but may in the worst case do double the work by visiting each state twice. This raises questions about the efficiency of the implementation of fairness.

5.3.1 The detection of SCCs

Table 5.4 shows the running costs associated with detecting SCCs in terms of the maximum stack depth and the running time in seconds. The models were first analysed for deadlock freedom with the SCC detection disabled (this is called *normal operation*), and then with SCC detection enabled, but without the use of the `goodchildren` flag.

There is not too significant a difference (a maximum of 9.7%) in execution times between the two approaches, suggesting that SCC detection is not expensive in terms of runtime. The only extra work that is required to detect SCCs, is the calculation of the $Lowlink(\cdot)$ function. This involves a single test and assignment per state, and an extra assignment that is performed in some states (those that satisfy the special condition discussed in Section 4.4). As during normal operation, each visited state is still pushed on and popped from the stack exactly once, although the removal of a state may be postponed until its entire SCC has been detected.

	Normal operation		*SCC detection*	
Model	*Depth*	*Time*	*Depth*	*Time*
DP8	27617	29.35	86154	31.98
EL4	2109	341.18	2325	374.55
PS2	10895	19.07	49606	19.19
SW3	2589	33.74	3972	36.25

Table 5.4: Overhead of SCC detection
(Time measurements are given in seconds)

Memory requirements, on the other hand, increase dramatically for some models. Initially it was feared that in all but a few cases the entire state graph would form a single SCC and reside on the stack for the duration of a verification run. In this case the space requirements of the stack would change from roughly logarithmic to strictly linear in the number of states. Fortunately, contrary to these expectations, most of the models consist of many small SCCs. In addition, a large number of single-state SCCs are formed for every model.

As Table 5.5 shows, in the case of EL4 and SW3 the size of the largest SCC is negligible relative to the size of the state graph, and little additional stack space is required. For DP8 and PS2 the opposite is true: the largest SCC is more than half the size of the state graph

for DP8 and the stack requirements increased more than threefold. All models investigated have fallen into one of these two classes (either small SCCs/small stack requirements or large SCCs/significant stack requirements). At this stage, no explanation for this is forthcoming and the phenomenon deserves attention. Further investigation may lead to techniques that reduce the space requirements.

Model	*Largest SCC as % of unique states*	*SCC:normal stack size ratio*
DP8	57.13	3.12
EL4	0.02	1.10
PS2	35.60	4.55
SW3	0.80	1.53

Table 5.5: SCC sizes and stack requirements

As explained in Section 4.2, the stack does not store state vectors, but only references to their positions in the cache. The stack contains only auxiliary information (for example, the `lowlink` field), and its size is independent of the size of the state vector. Moreover, for large models, the size of the stack is small compared to the size of the state cache.

5.3.2 Model checking overhead

The overhead associated with the operation of the `goodchildren` flag is more difficult to measure. It involves one test and a possible assignment (to propogate the `goodchildren` flag to the parent state) for every state that is removed from the stack, and one assignment each time the model checking algorithm finds that a state satisfies the CTL correctness specification. Space requirements are negligible: as explained in Section 4.4.4, a single pointer suffices.

Since this flag records the presence of satisfying paths in the state graph, its use depends on the particular nature of the model and the correctness specification. Different CTL specifications can lead to very different explorations of the same state graph. To gauge the influence of the implementation of fairness, some actual properties are checked. Table 5.6 gives the results of runs of the process scheduler model (PS2) with four different correctness specifications.

The first run checks for deadlock freedom, while the others deal with properties of variable

Property	*States*	*Transitions*	*Time*	*Result*
Deadlock detection	131688	576867	19.07	no deadlock
$AG(DeviceDriver.id < 3)$	131688	576867	23.20	satisfied
$AF(DeviceDriver.id = 2)$	73592	322571	12.87	satisfied
$AF(DeviceDriver.id = 3)$	49606	204577	8.00	violated

Table 5.6: Four verification runs of the process scheduler model (PS2)
(Time measurements are given in seconds)

`id` in process `DeviceDriver`. The second run checks a safety property (invariant) and does not require fairness (therefore the `goodchildren` flag is not used and SCCs are not detected). It claims that the value of variable `id` in process `DeviceDriver` is always less than 3. The difference in the times of the first two runs gives an indication of the overhead involved in checking that Boolean expression $DeviceDriver.id < 3$ is true in every state of the state graph.

The last two runs check liveness properties and require fairness. These give an indication of the performance of the model checker when checking meaningful specifications. As the third run illustrates, the system does not need to explore all states to establish the validity of these properties—it completed the analysis after exploring little more than half of the states. The fourth run terminated even quicker when a violation of the correctness specification was found. Moreover, the state graphs explored by these runs are not similar to the state graph of the first two runs. While checking liveness properties the exploration of some paths may be terminated early, leading to different revisits and a different structure for the state graph.

The most important observation is that the detection of fairness does not have a significant influence on the performance of the model checker. At least about 25000 transitions were analysed per second during all four runs shown above. (This is the standard rate, as is shown in Section 5.5.)

5.4 Interpretation

Although interpretation of programming languages is less efficient that native code generation, it is not clear cut that this is also true for state generation.

Since an implementation of a pre-compiled state generator is not available, and since there are significant differences between the new design and our older model checkers as well as the other model checkers like SPIN, we have to make an analytical instead of empirical comparison.

5.4.1 Interpreter v. pre-compiled state generator

The validator is driven by the on-the-fly CTL model checking algorithm, as explained in Chapter 3. As it explores the state graph, it repeatedly invokes procedure `Machine.Execute` to generate the next state. Each invocation of `Execute` involves the following steps (cf. Figure 4.2, page 49):

1. `Reschedule` is called to find a suitable transition;
2. `Step` interprets the transition's machine code; it may call
   (a) itself to evaluate expressions
   (b) `Trace.Change` and `Compact.UpdateValue` when updating variables
   (c) `Synchronise` to handle communication
3. `Trace.Update` marks which transition has been selected;
4. `Trace.Push` adds the new state to the stack.
5. `Trace.Change` and `Compact.UpdateValue` updates the location counter;

Steps 1 and 2 may be repeated a number of times while the selected transition involves a communication instruction that cannot synchronise in the current state. Such a transition is not truly enabled, but this can only be established by calling `Step`.

In a pre-compiled state generator steps 1 and 2 would be replaced with pre-compiled code, but steps 3–5 would remain the same. A pre-compiled state generator would still have to call `Trace.Change` and `Compact.UpdateValue` as in step 2(b) to update variables during assignments. Unless it maintains an uncompacted copy of the state, similar to the machine's variable store, it would also have to call `Compact.GetValue` every time the value of a variable was needed. Hence, the only procedures that truly belong to the interpreter are `Synchronise`, `Step` and `Reschedule`. In Table 5.7 an execution profile for a run of the model checker on the EL4 model is shown. When running times of the interpreter procedures are accumulated, the

interpreter runs for 52.4% of the time. For the other models, the total ranges from 49.3% to 64.4%.

	Module	*Procedure*	
1.5%	`Logic`	`CheckDeadlock`	1.5%
57.6%	`Machine`	`Synchronise`	3.7%
		`Step`	37.9%
		`Backtrack`	0.1%
		`Reschedule`	10.8%
		`Execute`	5.1%
14.3%	`Trace`	`Undo`	5.2%
		`Push`	2.6%
		`Pop`	1.6%
		`SetGoodChildren`	0.0%
		`Update`	1.2%
		`StepUpdate`	0.6%
		`Change`	3.1%
19.2%	`Cache`	`Clear`	0.3%
		`Hash1`	4.0%
		`Hash2`	2.1%
		`NewInsert`	11.0%
		`Lookup`	1.2%
		`SetFlag`	0.6%
6.8%	`Compact`	`Compare`	3.4%
		`Assign`	0.5%
		`UpdateValue`	2.9%

Table 5.7: Execution profile of the model checker for EL4
(Total running time is 341.18 seconds)

As the examples in Section 4.1.2 show, the fetching and decoding costs are small, and the real cost of interpretation lies in the execution of instructions. However, the actions of the interpreter while executing instructions are close to that of a pre-compiled state generator: variables are tested to see whether a transition is legal, and assignments are made to instantiate the transition. There are no extra procedure calls in the case of the interpreter.

Furthermore, a single transition such as the assignment `x := 1` is turned into several microtransitions: `pushAddress` $addr_{\texttt{x}}$, `pushValue` 1, `popVariable`. Fortunately, the microtransitions are "glued" together and executed within a single invocation of `Step`. In fact, this idea is taken

even further: ESML processes have no shared memory so that the only points of interference are communication commands and assignments that affect the truth value of the correctness specification. The interpreter can therefore carry out instructions until such an interference point or a rescheduling point (a point where processes should be given the opportunity to interleave) is reached. This implements the technique described in [58(Section 7.1)], and amounts to the "virtual coarsening of atomic actions" suggested in [46].

5.4.2 The composition of the instruction set

Table 5.8 shows the eight most frequently executed instructions for each of three models. The table indicates how many times each instruction is executed compared to other instructions. For example, during the analysis of DP9, 31.69% of executed instructions were `pushValue` instructions. Although this table gives no indication of the *cost* of each instruction, it does allow for some important observations.

DP9	
Instruction	*%*
`pushValue`	31.69
`end`	16.15
`bang`	11.36
`pushVariable`	10.40
`guards`	7.23
`index`	5.66
`add`	2.84
`mod`	2.81
Other	11.86

EL4	
Instruction	*%*
`pushValue`	30.24
`end`	12.74
`pushVariable`	9.51
`guards`	6.17
`pushParameter`	6.05
`hook`	6.02
`evaluate`	4.47
`popVariable`	4.47
Other	20.33

PS3	
Instruction	*%*
`pushValue`	23.95
`end`	17.72
`guards`	11.60
`hook`	11.14
`bang`	7.45
`poll`	7.17
`pushVariable`	4.18
`jump`	3.82
Other	12.97

Table 5.8: Instruction frequency for DP9, EL4 and PS3

The most frequent instructions by far are `pushValue` and `end`. Together they account for roughly two fifths of all executed instructions. Their implementations are relatively straightforward and leave little room for optimisation, but it is possible to optimise their use.

The `pushValue` instruction places a constant value on the stack. It is used in three situations: to generate constants during expression evaluation, to compute the addresses for variables, and

to compute record field offsets. Little can be done in the case of expression constants, but easy optimisations are possible in the other two cases:

1. The value of variable `x` is placed on the stack with `pushValue` $addr_{\texttt{x}}$, `pushVariable`. A new instruction `loadVariable` $addr_{\texttt{x}}$ could replace this combination. That this may be a prudent optimisation is suggested by the relatively high frequency of `pushVariable` instructions.

2. To place the address of record field `r.x` on the stack, the compiler generates the following instructions: `pushValue` $addr_{\texttt{r}}$, `pushValue` $offset_{\texttt{x}}$, `add`. The calculation of $addr_{\texttt{r}}$ could be arbitrarily complex (consider for instance `r[`*`expression`*`].x`), but the last two instructions could be replaced with an instruction `addOffset` $offset_{\texttt{x}}$. This change will have no effect on the models in this chapter, but it would of course benefit models that make use of records.

The `end` instruction terminates an expression subroutine that was invoked either explicitly by an `evaluate` instruction, or implicitly by another instruction that needs to evaluate an expression, such as a guarded command. In fact, almost all instructions invoke an expression subroutine. As explained in Section 4.1.2, the motivation behind these subroutines is the idea that if frequently used, they could be translated to the native instruction set to boost the performance of the model checker. As the tables above show, arithmetic instructions are not that common compared to other instructions. The increase in performance afforded by native expression instructions would not justify the effort. An easier and surer improvement is the total elimination of this feature. This would eliminate the `end` instruction and the associated procedure call.

The next most frequent instructions are `guards` and `pushVariable`. Both the guarded command iteration (`DO`) and selection (`IF`) commands are translated to a `guards` instruction. None of the models contain deadlock; all make use of nonterminating loops and this account for the high frequency of `guards` instructions. Moreover, every satisfied guard that is executed, is counted as an instance of a `guards` instruction.

Among the DP9 instructions, `add` and `mod` appear an equal number of times, while `index` is

twice as common as either of the first two instructions. This pattern is explained by the fact that DP9 uses an array data structure which is invariably accessed as `s[k]` and `s[(k+1) MOD m]`.

5.5 Overall performance

The overall performance of the model checker is assessed in a comparison with the SPIN system (version 2.9.7) [34]. SPIN differs from the system described in this thesis in a number of ways: most importantly, it uses the automata-theoretic approach to perform LTL model checking, but can also check exclusively for deadlock. SPIN supports no particular form of fairness; instead, a "second search" technique is used to verify liveness properties that are specified as part of the correctness specification [25]. Models are written in Promela, a modelling language for protocols [30]. Promela supports dynamic process creation, a rich set of control structures and synchronous as well as asynchronous, buffered communication between processes. Promela control structures include all those of ESML, except for the `POLL` command which cannot be expressed in Promela. On the other hand, the ESML type system contains fewer but more versatile data structures than that of SPIN.

Another, more important difference lies in the transition systems produced by Promela and ESML models: the difference in the ratio of transitions to unique states has already been noted in Section 5.2: ESML models usually result in state graphs with a higher number of transitions per state. Also, depth-first search paths in ESML models are usually deeper than those in Promela models. The comparison of the dining philosophers ESML model (DP9) and the Promela model of a flow control layer (*pftp*) is typical in this regard: both models have roughly the same size (in terms of unique states); for *pftp* the transition to state ratio is 2.96, and the deepest path is 5780 states long; for DP9 the transition to state ratio is 5.57, and the deepest path is 62280 states long.

Because of these differences, any comparison of the "same" model in both languages would be misleading. Instead, Table 5.9 gives performance figures for four Promela models (distributed with the SPIN system), three ESML models with roughly corresponding sizes, and a fourth, larger ESML model. The last column of Table 5.9 gives an indication of the memory requirements in megabytes.

Promela models				
Model	*States*	*Transitions*	*Transitions per second*	*Memory*
leader	45885	185032	19917	9.544
snoopy	91920	305460	24052	10.607
pftp	439895	1301624	18424	56.175
sort	>526031	>2661181	19677	70.204

ESML models				
Model	*States*	*Transitions*	*Transitions per second*	*Memory*
DP7	44544	203233	25060	1.551
SW2	100426	447608	32459	3.002
DP9	420096	2338593	22883	9.461
PS3	2016168	9908147	28519	42.132

Table 5.9: Comparison of Promela and ESML models
(Memory requirements are given in megabytes)

All models were analysed for deadlock freedom (*invalid endstates* in SPIN terminology). SPIN's exhaustive search was used: in this mode, states are stored in a table similar to the state cache, but no states are replaced and, when the table is full, the analysis of the model is aborted (as in the case of the *sort* model). Similarly, the ESML models were analysed with enough memory to ensure that no cache state replacements occur.

The model checker compares favourably to SPIN both in the number of transitions it is able to check per second as well as in the memory required to do so. In [57(Section 5.1.4)] SPIN is reported to be 30% faster than the previous version of the model checker; as the table shows, the new design can analyse roughly the same number of transitions per second as SPIN. Moreover, the new system requires less memory and is therefore able to analyse far larger models than the SPIN system.

The largest ESML model analysed to date is a description of the VMTP protocol [9]. It generated 6.67 million unique states and 28.69 million transitions, took 2283 seconds (elapsed time) on a SPARCserver 1000, and required 111.905 megabytes of memory.

5.6 Summary

In summary, this chapter contains the following results:

- The compaction technique reduces the size of the state vector and the memory requirements roughly 5 times, and decreases runtime by a factor of 0.54–0.80.
- The state cache behaviour conforms to that described in the literature: it is able to support a state space of twice its size without seriously affecting performance.
- The detection of SCCs has little effect on the runtime of the model checker. In some cases the additional memory requirements are negligible, but at other times memory requirements are doubled. Model checking realistic claims have little effect on the performance.
- The interpreter accounts for 49.3–64.4% of the runtime. It was argued that a pre-compiled state generator cannot avoid this overhead. The instruction set can be optimised in several small ways, but no major improvements are possible.
- Overall, the model checker compares favourably to the SPIN system. The systems can explore roughly the same number of states per second (in the order of 25000), but the ESML model checker requires only about half as much memory as SPIN for the same number of states. In 64 megabytes of memory about 2000000 states can be analysed without overwriting states in the cache or making use of swapping.

Chapter 6

Conclusion

This thesis examined the issues involved in providing efficient support for on-the-fly model checking algorithms. From the discussion in Chapter 3, the implementation described in Chapter 4 and the results in Chapter 5, the following picture emerges:

- *Generation of states*: We presented intuitive arguments that the use of an abstract machine is simpler and more reliable that pre-compiled transition systems. We also argued in Section 5.4 that in the context of model checking interpretation is not much less efficient than pre-compilation of transition systems. Although no direct comparison was possible, interpretation did not penalise the model checker to such an extent that it is significantly less efficient than a model checker such as SPIN.

- *Representation of states*: We presented a simple technique that yields compact states but is runtime efficient at the same time. More advanced compression techniques incur too high an overhead exactly when it is least affordable—when runtimes are long.

- *Detection of revisited states*: When used in combination with the state compaction technique, state caching can handle large state graphs efficiently, making it unnecessary to resort to bitstate hashing.

- *Fairness constraints*: Fairness is essential for checking liveness properties. As noted in Section 2.5, specifying fairness constraints as part of the correctness property is cumbersome and CTL is not even capable of expressing all major forms of fairness. We therefore

elected to implement native strong fairness and showed that it is runtime efficient. In some cases, a significant amount of extra storage is required for SCC detection, but this task is often performed as part of model checking CTL anyway. It is not clear which approach—builtin fairness, or fairness expressed as part of the correctness specification—is more efficient; this is open question and material for future work.

In its present state the model checker provides a valuable basis for exploring new ideas. The modular structure facilitates changes; this has already proven invaluable for conducting experiments and for making the modifications necessary for the measurements presented in Chapter 5.

Although they were not discussed in this thesis, partial order techniques are popular because they are simple to implement and can significantly reduce the number of states explored, thereby boosting the performance of the state cache. We have made a preliminary implementation of the technique in [23] and have obtained promising results.

Final thoughts

There can be little doubt that the use of formal methods is on the increase [11, 48]. The state explosion problem notwithstanding, there seems to be a trend towards the direct verification of large programs written in conventional programming languages such as Java and C. Rather than focus on demonstrating complete correctness, the goal is to improve the quality of software by detecting as many bugs or design flaws as possible, and integrating formal verification and testing techniques. At the same time, more advanced techniques for model checking continue to be developed.

Cynics may argue that this trend may cause programmers to become more complacent about rigorous thinking, but to serious developers of critical software this comes as good news.

Appendix A

Model source code

A.1 Model of seven dining philosophers (DP7)

```
1  MODEL DiningPhil7;
2  CONST
3    max = 7;
4  TYPE
5    int = 0..max;
6    prot = {down(int), up(int)};
7    sticks = ARRAY [max] OF BOOLEAN;
8  VAR
9    p: prot;
10
11 PROCESS Chopstick(IN p: prot);
12 VAR s: sticks; k: int;
13 BEGIN
14   k := 0;
15   DO k < max ->
16     s[k] := TRUE; k := k + 1
17   END;
18   s[0] := FALSE;
19   s[1] := FALSE;
20
21   DO TRUE ->
22     POLL p?up(k) & (s[k] & s[(k + 1) MOD max]) ->
23         s[k] := FALSE;
24         s[(k + 1) MOD max] := FALSE
25     [] p?down(k) & ~(s[k] OR s[(k + 1) MOD max]) ->
26         s[k] := TRUE;
27         s[(k + 1) MOD max] := TRUE
28     END
29   END
30 END Chopstick;
31
32 PROCESS Phil0(OUT p: prot);
33 BEGIN
34   DO TRUE ->
35     (* eat *)
36     p!down(0);
37     (* meditate *)
38     p!up(0)
39   END
```

```
40 END Phil0;
41
42 PROCESS Phil(OUT p: prot; nr: int);
43 BEGIN
44   DO TRUE ->
45     (* meditate *)
46     p!up(nr);
47     (* eat *)
48     p!down(nr)
49   END
50 END Phil;
51
52 BEGIN
53   Chopstick(p);
54   Phil0(p);
55   Phil(p, 1); Phil(p, 2); Phil(p, 3);
56   Phil(p, 4); Phil(p, 5); Phil(p, 6)
57 END DiningPhil7
```

A.2 Model of the elevator with three floors (EL3)

```
1  MODEL Elevator3;
2  CONST
3    floors = 3; max = 5;
4  TYPE
5    int = 0..max - 1;
6    state = resting, goingup, goingdn;
7    random = {value(int)};
8    control = {dest(int), opendoor(int), getoff(int)};
9    lift = {upat(int), dnat(int)};
10   button = {up(int), dn(int), goto(int)};
11   queue = ARRAY [floors + 1] OF int;
12 VAR
13   r: random; c: control; l: lift; b: button;
14
15 PROCESS Random(OUT r: random);
16 VAR seed: int;
17 BEGIN
18   DO TRUE -> seed := (3 * seed + 4) MOD max; r!value(seed) END
19 END Random;
20
21 PROCESS Person(floor: int; IN r: random; IN c: control; OUT b: button);
22 VAR x: int;
23 BEGIN
24   DO TRUE -> x := floor;
25     DO x = floor -> r?value(x); x := x MOD floor + 1 END;
26     IF x < floor -> b!dn(floor) [] x > floor -> b!up(floor) END;
27     c?opendoor(floor);
28     b!goto(x);
29     c?getoff(x)
30   END
31 END Person;
32
33 PROCESS Lift(IN c: control; OUT l: lift);
34 VAR s: state; f, d: int;
35 BEGIN
36   f := 1; s := resting;
37   DO s = resting ->
38     c?dest(d);
39     IF d < f -> s := goingdn [] d >= f -> s := goingup END
```

```
40    [] s = goingup ->
41      c?dest(d);
42      IF f = d -> l!upat(f); s := resting
43      [] f # d -> f := f + 1
44      END
45    [] s = goingdn ->
46      c?dest(d);
47      IF f = d -> l!dnat(f); s := resting
48      [] f # d -> f := f - 1
49      END
50    END
51  END Lift;
52
53  PROCESS Control(IN b: button; OUT c: control; IN l: lift);
54  VAR uq, dq, out: queue; d, f: int;
55  BEGIN
56    DO TRUE ->
57      POLL b?up(f) -> uq[f] := uq[f] + 1
58      [] b?dn(f) -> dq[f] := dq[f] + 1
59      [] l?upat(f) ->
60        DO uq[f]>0 ->
61          c!opendoor(f);
62          b?goto(d); out[d] := out[d] + 1;
63          uq[f] := uq[f] - 1
64        END;
65        DO out[f]>0 ->
66          c!getoff(f);
67          out[f] := out[f] - 1
68        END;
69        d := f + 1;
70        DO (d <= floors) & ~((uq[d] > 0) OR (out[d] > 0)) -> d := d + 1 END;
71        IF d > floors -> d := f - 1;
72          DO (d > 0) & ~((dq[d] > 0) OR (out[d] > 0)) -> d := d - 1 END
73        END
74      [] l?dnat(f) ->
75        DO dq[f] > 0 ->
76          c!opendoor(f);
77          b?goto(d);
78          out[d] := out[d] + 1;
79          dq[f] := dq[f] - 1
80        END;
81        DO out[f] > 0 ->
```

```
82          c!getoff(f);
83          out[f] := out[f] - 1
84        END;
85        d := f - 1;
86        DO (d > 0) & ~((dq[d] > 0) OR (out[d] > 0)) -> d := d - 1 END;
87        IF d = 0 -> d := f + 1;
88          DO (d <= floors) & ~((uq[d] > 0) OR (out[d] > 0)) -> d := d + 1 END;
89          IF d > floors -> d := 0 [] d <= floors -> SKIP END
90        END
91      [] c!dest(d) & (d # 0) -> SKIP
92      END
93    END
94  END Control;
95
96  BEGIN
97    Random(r);
98    Person(1, r, c, b);
99    Person(2, r, c, b);
100   Person(3, r, c, b);
101   Control(b, c, l);
102   Lift(c, l)
103 END Elevator3
```

A.3 Model of the process scheduler with one process (PS1)

```
1  MODEL Scheduler1;
2  CONST
3    nullproc = 0; qmax = 2;
4  TYPE
5    procid = 0..2;
6    readyrequest = {selectproc(procid), enterproc(procid)};
7    runningrequest = {newproc(procid), kcall(procid), int, tick};
8    iocommand = {doio(procid), iocomplete};
9    devicecommand = {startio};
10   continue = {resume(procid)};
11   procqueue = LIST [qmax] OF procid;
12 VAR
13   ch0: readyrequest;
14   ch1: runningrequest;
15   ch2: continue;
16   ch3: iocommand;
17   ch4: devicecommand;
18
19 PROCESS User(id: procid; OUT run: runningrequest; OUT rr: readyrequest;
20   IN c: continue);
21 VAR p: procid;
22 BEGIN
23   rr!enterproc(id);
24   DO TRUE ->
25     POLL c?resume(p) & p = id -> SKIP END;
26     run!kcall(id)
27   END
28 END User;
29
30 PROCESS Timer(OUT run: runningrequest);
31 BEGIN
32   DO TRUE -> run!tick END
33 END Timer;
34
35 PROCESS Ready(IN rr: readyrequest; OUT run: runningrequest);
36 VAR p: procid; q: procqueue;
37 BEGIN
38   p := nullproc; EMPTY(q);
39   DO TRUE ->
```

```
40      POLL rr?selectproc(p) ->
41        IF LEN(q) > 0 -> p := HEAD(q); REMOVE(q); run!newproc(p)
42        [] LEN(q) = 0 -> run!newproc(nullproc)
43        END
44      [] rr?enterproc(p) ->
45        IF p # nullproc -> APPEND(q, p)
46        [] p = nullproc -> SKIP
47        END
48      END
49    END
50  END Ready;
51
52  PROCESS Running(OUT rr: readyrequest; IN run: runningrequest;
53    OUT io: iocommand; OUT c: continue);
54  TYPE procstate = 0..1;
55  VAR curproc: procid; state: procstate;
56  BEGIN
57    rr!selectproc(nullproc); run?newproc(curproc);
58    IF curproc # nullproc -> c!resume(curproc)
59    [] curproc = nullproc -> SKIP
60    END;
61    DO TRUE ->
62      POLL run?kcall(curproc)-> io!doio(curproc)
63      [] run?int -> io!iocomplete
64      [] run?tick -> SKIP
65      END;
66      rr!selectproc(curproc); run?newproc(curproc);
67      IF curproc # nullproc -> c!resume(curproc)
68      [] curproc = nullproc -> SKIP
69      END
70    END
71  END Running;
72
73  PROCESS DeviceDriver(OUT rr: readyrequest; IN io: iocommand;
74    OUT dc: devicecommand);
75  VAR curproc, id: procid; rq: procqueue; idle: BOOLEAN;
76  BEGIN
77    EMPTY(rq); idle := TRUE;
78    DO TRUE ->
79      POLL io?doio(id) & idle -> curproc := id; dc!startio; idle := FALSE
80      [] io?doio(id) & ~idle -> APPEND(rq, id)
81      [] io?iocomplete ->
```

```
82        rr!enterproc(curproc);
83        IF LEN(rq) = 0 -> idle := TRUE
84        [] LEN(rq) > 0 -> curproc := HEAD(rq); REMOVE(rq); dc!startio
85        END
86      END
87     END
88  END DeviceDriver;
89
90  PROCESS Device(OUT rr: runningrequest; IN dc: devicecommand);
91  VAR idle: BOOLEAN;
92  BEGIN
93    idle := TRUE;
94    DO TRUE ->
95      POLL dc?startio & idle -> idle := FALSE END;
96      idle := TRUE; rr!int
97    END
98  END Device;
99
100 BEGIN
101   Timer(ch1); User(1, ch1, ch0, ch2);
102   Ready(ch0, ch1); Running(ch0, ch1, ch3, ch2);
103   DeviceDriver(ch0, ch3, ch4); Device(ch1, ch4)
104 END Scheduler1
```

A.4 Model of the sliding window protocol with window size one (SW1)

```
1  MODEL SlidingWindow1;
2  CONST
3    N = 1; M = 2*N;   (* N is the window size *)
4  TYPE
5    int = 0..M;
6    set = ARRAY [N] OF BOOLEAN;
7    C = {org, msg(Int), set(Set)};
8  VAR
9    in, out, sf, fr, rb, bs: C;
10
11 PROCESS Forward(IN sf: C; OUT fr: C);
12 VAR n: int;
13 BEGIN
14   sf?msg(n);
15   DO TRUE -> sf?msg(n)
16   [] TRUE -> fr!msg(n)
17   END
18 END Forward;
19
20 PROCESS Backward(IN rb: C; OUT bs: C);
21 VAR s: set;
22 BEGIN
23   rb?set(s);
24   DO TRUE -> rb?set(s)
25   [] TRUE -> bs!set(s)
26   END
27 END Backward;
28
29 PROCESS Send(IN in, bs: C; OUT sf: C);
30 VAR n, m, x: int; s: set;
31 BEGIN
32   x := 0;
33   DO x < N ->
34     s[x] := TRUE; x := x + 1
35   END;
36   m := 0; n := 0;
37   DO TRUE ->
```

```
38      POLL in?org & (n # m + N) ->
39        n := (n + 1) MOD M
40      [] sf!msg(x) & (x < n) ->
41        SKIP
42      [] bs?set(s) ->
43        DO ~s[m MOD N] & (m < n) ->
44          m := (m + 1) MOD M
45        END
46      END;
47      x := m;
48      DO ~s[x MOD N] & (x < n) ->
49        x := (x + 1) MOD M
50      END
51    END
52  END Send;
53
54  PROCESS Receive(IN fr: C; OUT rb, out: C);
55  VAR n, x: int; s: set;
56  BEGIN
57    x := 0;
58    DO x < N ->
59      s[x] := FALSE; x := x + 1
60    END;
61    n := 0;
62    DO TRUE ->
63      POLL out!org & s[n MOD N] ->
64        s[n MOD N] := FALSE;
65        n := (n + 1) MOD M
66      [] fr?msg(x) ->
67        IF x >= n -> s[x MOD N] := TRUE
68        [] x < n -> SKIP
69        END
70      [] rb!set(s) & ~s[n MOD N] ->
71        SKIP
72      END
73    END
74  END Receive;
75
76  PROCESS Source(OUT in: C);
77  BEGIN
78    DO TRUE -> in!org END
79  END Source;
```

```
80
81  PROCESS Sink(IN out: C);
82  BEGIN
83    DO TRUE -> out?org END
84  END Sink;
85
86  BEGIN
87    Source(in);  Send(in, bs, sf);      Forward(sf, fr);
88    Sink(out);   Receive(fr, rb, out);  Backward(rb, bs)
89  END SlidingWindow1
```

Appendix B

Model analysis details

Details of the experiments described in Chapter 5 are given below. Sections B.1–B.4 contain information about the four sets of models: DPn (dining philosophers), ELn (elevator), PSn (process scheduler) and SWn (sliding window).

As explained in Chapter 5, tests were conducted on an SGI Indy workstation with a 150MHz MIPS R4400 processor and 64 megabytes of physical memory. Swapping was not disabled, but tests were run with a minimal system load to guarantee access to as much physical memory as possible. Time was measured using the standard Unix `getrusage` system call, that returns information about the use of resources. This includes the time spent executing the process and its system calls (excluding time waiting for IO request completion).

Each table contains the following information:

- *Unique* is the number of unique states in the state graph. *Revisits* is the number of states that were revisited (i.e. reached more than once). *Loops* is the number of cycles detected in the state graph. *Transitions* is the number of transitions executed and is equal to the sum of the first three fields. The *Transitions/state* ratio gives an indication of the number of revisits per state.
- *Time* is the time in seconds required to check for deadlock freedom with state compaction. *Transitions/second* is the number of transitions checked per second.

- *Bits* is the size of the compacted state vector. The fraction of the potential state space that is reached is $Unique/2^{Bits}$. The minimum number of bits required to represent the unique states is $\lceil \log_2 Unique \rceil$. *Processes* is the number of processes in the model.

- *Cache size* is given in the form 230K+1, meaning that memory is reserved for 230001 cache entries (here *K* denotes a unit of 1000 entries). *Probes* is the maximum number of probes necessary to find a state in the cache. *Depth* is the length of the longest path explored. *Delta* is the maximum number of (16 bit) words required for delta storage. *Memory* is given in megabytes and includes the requirements of the cache, stack and delta storage.

- **Without compaction:** *Time* is the time in seconds required to check for deadlock freedom without state compaction. *Transitions/second* is the number of transitions checked per second. *Bits* is the size of the uncompacted state vector. The memory in megabytes required for the state cache, stack and delta storage is *Memory*. The ratio of the compacted and uncompacted times is the *Slowdown* factor. In the case of EL4, PS3 and SW9 is was not possible to perform these analyses, since the memory required exceeded that of the workstation.

  The number of `Compare` and `UpdateValue` operations remain the same as when compaction is active, but are given as an estimate of the extra work. The number of `Assign` operations is equal to the number of unique states, as explained in Section 5.1.1.

- **With SCC detection:** *Time* is the time in seconds required to check for deadlock freedom with state compaction and SCC detection activated. The memory in megabytes required for the state cache, stack and delta storage is *Memory*. *Largest* is the number of states in the largest SCC, and *Number* is the number of different SCCs. *Depth* is the maximum number of states on the stack during the analysis. The space required for delta storage is the same as before, since the storage of undo information is not affected by SCC detection.

B.1 Measurements for dining philosophers models (DPn)

Property	*DP7*	*DP8*	*DP9*
Unique	44544	137664	420096
Revisits	116916	395453	1342591
Loops	41773	164132	575906
Transitions	203233	697249	2338593
Transitions/state	4.56	5.06	5.57
Time	8.11	29.35	102.20
Transitions/second	25060	23756	22883
Bits	50	54	58
Unique/2^{Bits}	$4{\times}10^{-11}$	$8{\times}10^{-12}$	$2{\times}10^{-12}$
$\lceil \log_2 \text{Unique} \rceil$	16	18	19
Processes	8	9	10
Cache size	100K+1	200K+3	600K+1
Probes	8	23	24
Depth	9109	27617	62280
Delta	34504	104602	236058
Memory	1.551	3.423	9.461
Without compaction			
Time	10.19	36.83	133.52
Transitions/second	19950	18933	17515
Bits	272	304	336
Memory	4.282	9.673	30.555
Slowdown	1.26	1.25	1.31
`Compare`	223812	1027369	3540667
`UpdateValue`	301920	1033632	3459168
With SCC detection			
Time	8.86	31.98	114.45
Memory	2.132	5.166	15.836
Largest	24344	78653	207178
Number	583	10626	24583
Depth	27329	86154	266184

B.2 Measurements for elevator models (ELn)

Property	*EL3*	*EL4*
Unique	68193	1633032
Revisits	210585	6179698
Loops	44341	1273869
Transitions	323839	9086599
Transitions/state	4.75	5.56
Time	10.76	341.18
Transitions/second	30094	26633
Bits	89	107
Unique/2^{Bits}	$1{\times}10^{-22}$	$1{\times}10^{-26}$
$\lceil \log_2 \text{Unique} \rceil$	17	21
Processes	6	7
Cache size	150K+1	2250K+1
Probes	12	35
Depth	856	2109
Delta	3070	7578
Memory	2.381	44.053
Without compaction		
Time	19.78	·
Transitions/second	16376	·
Bits	432	512
Memory	7.069	149.522
Slowdown	1.84	·
`Compare`	390948	18406626
`UpdateValue`	483252	13640289
With SCC detection		
Time	10.74	374.55
Memory	2.381	44.053
Largest	405	405
Number	21221	605918
Depth	997	2325

B.3 Measurements for process scheduler models (PSn)

Property	*PS1*	*PS2*	*PS3*
Unique	8456	131688	2016168
Revisits	19211	377595	6933306
Loops	4448	67584	958673
Transitions	32115	576867	9908147
Transitions/state	3.80	4.38	4.91
Time	1.13	19.07	347.43
Transitions/second	28420	30256	28519
Bits	59	69	78
Unique/2^{Bits}	$1{\times}10^{-14}$	$2{\times}10^{-16}$	$7{\times}10^{-18}$
$\lceil \log_2 \text{Unique} \rceil$	14	18	21
Processes	6	7	8
Cache size	100K+1	200K+3	2500K+1
Probes	3	22	52
Depth	1358	10895	80729
Delta	4515	36785	275913
Memory	1.242	3.540	42.132
Without compaction			
Time	1.70	26.25	·
Transitions/second	18902	21978	·
Bits	320	352	416
Memory	4.367	9.790	139.788
Slowdown	1.50	1.38	·
`Compare`	29061	1014524	21462052
`UpdateValue`	46034	803842	13713588
With SCC detection			
Time	1.23	19.19	364.92
Memory	1.305	4.759	54.726
Largest	3451	46894	468544
Number	3788	71541	1362232
Depth	3865	49606	483699

B.4 Measurements for sliding window protocol models (SWn)

Property	*SW1*	*SW2*	*SW3*	*SW9*
Unique	22464	100426	235098	2673976
Revisits	61080	286649	681441	7974994
Loops	14905	60533	135903	1401971
Transitions	98449	447608	1052442	12050941
Transitions/state	4.38	4.48	4.48	4.51
Time	3.15	13.79	33.74	471.89
Transitions/second	31254	32459	31193	25538
Bits	43	53	56	89
Unique/2^{Bits}	3×10^{-9}	1×10^{-11}	3×10^{-12}	4×10^{-21}
$\lceil \log_2 \text{Unique} \rceil$	15	17	18	22
Processes	6	6	6	6
Cache size	250K+7	250K+7	350K+3	3500K+17
Probes	3	10	25	52
Depth	665	1791	2589	9561
Delta	2004	5456	7887	29025
Memory	2.965	3.002	4.209	55.057
Without compaction				
Time	4.71	20.38	54.35	·
Transitions/second	20907	21966	19365	·
Bits	240	288	336	624
Memory	8.825	9.838	16.514	287.480
Slowdown	1.50	1.48	1.61	·
`Compare`	90373	539143	1679839	25313655
`UpdateValue`	121998	563375	1341667	16382794
With SCC detection				
Time	3.41	14.92	36.25	446.47
Memory	2.965	3.033	4.241	55.276
Largest	564	1222	1926	8124
Number	1479	26033	66017	887165
Depth	825	2486	3972	16842

Appendix C

Abstract instruction set

The following sections document the instruction set of the abstract machine. The meaning of each instruction is given in a pseudo-code notation that refers to the machine's stack as s[] and to the stack pointer as t. The top element is s[t], the next to top element s[t−1], and so on, and the stack grows upwards from 0. The variable frame of the current process is referred to as v[] and its parameters as p[]. The $\perp$ sign represents a special value that denotes "undefined".

C.1 Arithmetic instructions

`mul`	s[t−1] := s[t−1] × s[t]; decrement t
`div`	Check that s[t] ≠ 0; s[t−1] := s[t−1] div s[t]; decrement t
`mod`	Check that s[t] ≠ 0; s[t−1] := s[t−1] mod s[t]; decrement t
`chs`	s[t] := −s[t]
`compare` n	s[t−2n+1] := 1 if s[t...t−n+1] = s[t−n...t−2n+1] elementwise, otherwise 0; decrement t by $2n-1$
`equal`	s[t−1] := 1 if s[t] = s[t−1], otherwise 0; decrement t
`neq`	s[t−1] := 1 if s[t] ≠ s[t−1], otherwise 0; decrement t
`greater`	s[t−1] := 1 if s[t] > s[t−1], otherwise 0; decrement t
`geq`	s[t−1] := 1 if s[t] ≥ s[t−1], otherwise 0; decrement t
`less`	s[t−1] := 1 if s[t] < s[t−1], otherwise 0; decrement t
`leq`	s[t−1] := 1 if s[t] ≤ s[t−1], otherwise 0; decrement t
`and`	s[t−1] := 1 if s[t] = s[t−1] = 1, otherwise 0; decrement t
`or`	s[t−1] := 1 if s[t] = 1 ∨ s[t−1] = 1, otherwise 0; decrement t

`not`	s[t−1] := 1 if s[t] = 0, otherwise 0; decrement t
`evaluate` a	Evaluate the expression subroutine at address a; the instructions at address a will place the result on the stack
`end`	Return to the caller of the expression subroutine

C.2 Memory manipulation instructions

`pushValue` x	increment t; s[t] := x
`pushVariable`	s[t] := v[s[t]]
`pushVariableRange` n	s[t...t+n−1] := v[s[t]...s[t]+n−1]; increment t by $n-1$
`popVariable`	v[s[t−1]] := s[t]; decrement t by 2
`popVariableRange` n	v[s[t]...s[t]−n+1] := s[t−1...t−n]; decrement t by $n+1$
`pushParameter`	s[t] := p[s[t]]
`pushParameterRange` n	s[t...t+n−1] := p[s[t]...s[t]+n−1]; increment t by $n-1$

C.3 List manipulation instructions

`length`	s[t] := length of list at v[s[t]]
`head` e	s[t...t+e−1] := the e-word head element of the list at v[s[t]]; increment t by $e-1$
`empty` n	Empty the n-word list at v[s[t]]; decrement t
`append` e	Append s[t−1...t−e] to the list at v[s[t]]; decrement t by $e+1$
`prepend` e	Prepend s[t−1...t−e] to the list at v[s[t]]; decrement t by $e+1$
`remove` e	Remove the e-word head element of the list at v[s[t]]; decrement t

C.4 Communication instructions

`bang` ch s a n c	Send signal s over channel ch; if $a \neq 0$, the n-word result of expression subroutine a is send along as data; if $c \neq \perp$, expression subroutine c must evaluate to 1 for the instruction to succeed
`hook` ch s a n c	Receive signal s over channel ch; if $a \neq 0$, the result of expression subroutine a is the address where n words of received data is stored; if $c \neq \perp$, expression subroutine c must evaluate to 1 for the instruction to succeed
`poll` a	Evaluate the guarded communication commands up to address a

C.5 Control flow instructions

`activate` n a v p ch	Activate the process with name n located at code address a; v and p are the size of the variable frame and the parameters, respectively; ch is the number of channel parameters
`terminate`	Terminate the current process
`guards` a	Evaluate the guarded commands up to address a
`jump` a	Jump to address a
`skip`	Advance to the next instruction
`trap`	Report an execution failure

C.6 Miscellaneous instructions

`outString`	Display the string of characters $\mathtt{s}[\mathtt{t}\ldots\mathtt{t}-k+1]$, where $\mathtt{s}[\mathtt{t}-k]=0$; decrement `t` by $k+1$
`outValue`	Display $\mathtt{s}[\mathtt{t}]$; decrement `t`
`outRange` n	Display $\mathtt{s}[\mathtt{t}\ldots\mathtt{t}-n+1]$; decrement `t` by n
`outLn`	Start a new line
`index` b m	Check that $\mathtt{s}[\mathtt{t}]<m$; $\mathtt{s}[\mathtt{t}-1]:=\mathtt{s}[\mathtt{t}-1]+b\times\mathtt{s}[\mathtt{t}]$; decrement `t`
`selectProc` n	Select the variable frame of process n

Appendix D

The ESML modelling language

ESML (Extended State Machine Language) is a high-level specification language designed for the modelling of reactive systems [14]. The design of ESML was inspired by CSP [29], Joyce [7, 8], and Promela [30]. Joyce is a strongly typed, concurrent programming language for distributed systems and is based on CSP and Pascal. Promela is a high-level specification language originally designed for protocol specification and used in the SPIN system [34].

ESML has the following important properties:

- Complex data structures such as records, arrays, and queues are intrinsic objects in the language.
- Concurrent processes that communicate via synchronous message passing are supported. Since communication instructions were identified as a common source of errors in [7], messages and communication channels are strongly typed.
- Dijkstra guarded command-style control structures with non-deterministic choice offer a mechanism for control flow abstraction.

D.1 Constant, type and variable definitions

Constants, user types and variables are defined following the `CONST`, `TYPE` and `VAR` keywords, respectively. The scope of an identifier extends from the end of its definition to the end of the block in which it is defined. Its block can either be the entire model, or a single process.

Constants

ESML allows the definition of integer and Boolean constant expressions (i.e., expressions that can be evaluated at compile time), following the keyword `CONST`. There are two predefined Boolean constants, `TRUE` and `FALSE`.

```
CONST
  windowsize = 5;
  maxmsgnumber = 2 * (windowsize + 1);
  lossychannel = TRUE;
  bigmodel = lossychannel OR (maxmsgnumber >= 20);
```

Basic types

Instead of a set of predefined types ESML offers only the `BOOLEAN` primitive type. New types can be defined by the user after the keyword `TYPE`.

Integer types are constructed using the subrange type construction. Subranges are specified by giving a lower and upper bound. This may seem restrictive, but it is easy to use and allows the encoding of assumptions about the ranges of variables. Assignments to variables are checked during the analysis of models to ensure that a variable is not assigned a value outside its range. In addition, this mechanism leads to smaller states. Enumeration types provide a symbolic set of values.

```
CONST
  maxprocess = 10;
TYPE
  processnumber = 0..maxprocess-1;
  processstate = running, ready, blocked, zombie;
```

Although the values of integer variables are restricted by their type, integer expressions may assume any value, so that the following assignment to a variable of type `processnumber` is legal:

```
p := (123 * p) MOD maxprocess;
```

Variables of different subrange types may be mixed in the same expression, as long as the resulting value is a valid value in the subrange of the variable to which it is assigned.

Structured types

More complex data types are defined with tuple, array or list constructions. These allow the grouping of related data, the mapping of integers to other data, and the modelling of queues.

```
CONST
  maxbitset = 16;
TYPE
  processrecord = (
    id, parentid: processnumber;
    state: processstate );
  bitset = ARRAY [maxbitset] OF BOOLEAN;
  processqueue = LIST [10] OF processrecord;
```

Tuple fields are accessed as ⟨*variable access*⟩`.`⟨*field identifier*⟩. Tuple field may be of any type, including other tuple types, but excluding alphabets types defined in the next section. Nested *definitions* (i.e., anonymous tuples) are not allowed. Entire tuples may be assigned to each other, but not to variables of another tuple type.

An array is indexed with integers in the range $0 \ldots n-1$, where n is the declared size of the array; index range checking is performed during the analysis. List contents are manipulated using list operations described below.

An array or list can have any base type including other arrays or lists, but excluding alphabet types. As with tuples, the base type must be an identifier: definitions of the form `ARRAY[`k`] OF ARRAY [`j`] OF` T are not allowed. The sizes of arrays and lists must be non-zero positive integer constants.

Alphabet types

The messages that are sent between processes are defined by *alphabet types*. Communication channels too are typed and may only carry messages of an appropriate type. An alphabet is a set of messages $\{m_0, m_1, \ldots, m_n\}$. Each message consists of a signal s_i and optionally an accompanying data value of type t_i, written $s_i(t_i)$. In the following example `ack` and `nak` are signals without data, while the `schedule` message carries a value of type `processrecord`.

```
TYPE
  protocol = { schedule(processrecord), ack, nak };
```

D.2 Expressions

Arithmetic based on the usual operations *addition* ("`+`"), *subtraction* ("`-`"), *multiplication* ("`*`"), *integer division* (`DIV`), *modulo* (`MOD`), and *unary negation* ("`-`"), is supported as is Boolean expressions with *conjunction* ("`&`"), *disjunction* (`OR`), and *Boolean negation* ("`~`"), and relational operators ("`=`", "`#`", "`<`", "`<=`", "`>`", "`>=`"). Relational operators return a `BOOLEAN` result. Parenthesis ("`(`", "`)`") can be used to group subexpressions appropriately.

Two special operators are defined for list types: `LEN(`*list*`)` returns the length of *list* and `HEAD(`*list*`)` returns the first element of *list* without removing it from the list.

Operators have the following precedence:

6	`( )`
5	`LEN HEAD`
4	`~` -(unary)
3	`& * DIV MOD`
2	`OR +` -(binary)
1	`= # < <= > >=`

Operators with higher precedence are evaluated first. Operators with the same precedence are evaluated in the order that they appear in an expression. Short circuit evaluation of Boolean expressions is guaranteed by the language.

D.3 Commands

The state of a model is changed by executing commands.

The simplest command is `SKIP` which has no effect other than advancing the location counter. However, it plays an important role in ESML, since empty sequences of commands are not allowed. This makes models easier to read and prevents unintentional omission of commands in the specification.

Assignment and list commands

Values are assigned to variables with the "`:=`" command. Before a value is stored in an integer variable, it is checked to ensure that value is valid for the variable's range. The left-hand side of the assignment must be a valid variable access as defined by the grammar (see Section D.5).

```
x := (x + 1) MOD 7
b[2] := b[1] * b[0]
process.state := running
```

It is permitted to assign an expression to a variable of the same type, even entire tuples, lists and arrays. However, assignment to communication channel variables is not allowed.

There are four special list commands: `EMPTY(`*list*`)` discards the contents of *list* and sets length to 0, `REMOVE(`*list*`)` discards the item at the head of the list, and `PREPEND(`*list*`,` x`)` and `APPEND(`*list*`,` x`)` insert the value x at the head and tail of the list, respectively.

Communication commands

Processes exchange messages by means of communication commands that operate on communication channels. There are two commands that perform communication: `!` (send) and `?` (receive). Signal `s` and the value of expression `e` are sent on channel `ch` by the command `ch!m(e)`. Similarly, signal `s` is received on channel `ch` and its associated data stored in variable `v` by the command `ch?m(v)`. A channel parameter that is marked with the `OUT` keyword cannot be used for send commands; similarly, a channel parameter marked with the `IN` keyword can

only be used for receive commands.

A pair of send and receive commands synchronise when they are both ready to execute and communicate the same signal over the same channel. Communication is blocking and a process must wait until a communication partner is found and the message transferred, before it can proceed.

The first command in the following example receives a `data` message and stores the accompanying value in the third element of array `x`, the second accepts the signal `ack`, and the third command sends a `setmsg` message with the value of `msgnr + 1`.

```
in?data(x[2])
in?ack
out!setmsg(msgnr + 1)
```

Note that the names of messages must be constants; the following code is not legal ESML:

```
v := nak;
out!v
```

Control structures

The ESML control structures are patterned after Dijkstra guarded commands [16]. Each control structure contains a list of $G \longrightarrow A$ pairs, where the guard G is a Boolean expression and the action A is a sequence of commands that is executed only if the guard is satisfied. ESML supports the following three constructs:

- `IF`: All guards are evaluated, one true guard is selected non-deterministically, and its action is executed. At least one `IF` guard must be true; if this is not true, the analysis of the model is aborted and the error is reported to the user.
- `DO`: All guards are evaluated, one true guard is selected non-deterministically, and its action is executed. This process is repeated until all guards are false.
- `POLL`: The guard–action pairs of this structure have a special form: $(C \ \& \ G) \longrightarrow A$, where G and A are the same as above and C is a communication command. If the communication command is a send (`!`), can execute and G evaluates true *before* the

execution of the command, the guard is satisfied. If the communication command is a receive (?), can execute and G evaluates to true *after* the command, the guard is satisfied. The DO blocks until one or more guards is satisfied, at which point one of the satisfied guards is selected non-deterministically, and its action is executed. This command does not repeat like the DO command.

The following IF command increments x if it is odd, or halves it when it is even:

```
IF x MOD 2 = 1 -> x := x + 1
[] x MOD 2 = 0 -> x := x DIV 2
END
```

The following example illustrates the use of non-determinism. The first guard of the DO construct checks that the queue q is non-empty; if so, the first element of the queue is removed and sent via channel out. The second guard is satisfied when the queue is not full; a new element is received via channel in and appended to the queue. While the queue is neither full nor empty, a non-deterministic choice is made.

```
DO
   LEN(q) > 0 ->
      x := HEAD(q);
      REMOVE(q);
      out!send(x)

[] LEN(q) < max ->
      in?recv(x);
      APPEND(q, x)
END
```

Because reactive processes are not supposed to terminate the following construction is common in ESML models:

```
DO TRUE ->
  (* receive and react on messages from the environment *)
END
```

The last example illustrates the use of the POLL command. The construct is ready to accept the reset signal via the control channel, or to send the val message with the value of local

variable `n` via the `client` channel, unconditionally. It accepts the `inc` signal via channel `client` only if `n` is less than `max`, and the `dec` signal via channel `client` only if `n` is greater than 0.

```
POLL
   control?reset      -> n := 0
[] client!val(n)      -> SKIP
[] client?inc & n<max -> n := n + 1
[] client?dec & n>0   -> n := n - 1
END
```

Process activation

New processes are created by an activation command similar to procedure invocation. The actual arguments passed to the process must must match the formal parameters in number and type. During activation, storage is allocated for the local variables.

```
Producer(in);
Consumer(out);
Buffer(in, out, 10, FALSE)
```

Trace commands

The `TRACE` command are useful when developing models. Such a command displays the values its arguments. It can take an arbitrary number of arguments, including string constants. Non-integer values are mapped to integer values: `FALSE` maps to 0, and `TRUE` to 1. Members of an enumeration type is numbered sequentially with the first member numbered 0; these numbers are printed by the `TRACE` command when displaying a variable of this type.

```
TRACE("process id=", pr[x+1].id)
```

D.4 Processes and models

Processes have the following structure:

```
PROCESS name (parameters);
  CONST constant definitions
  TYPE type definitions
```

```
  VAR variable definitions
BEGIN
  commands
END name;
```

The constant, type and variable definitions are optional, but, when present, are local to the process and not visible to other processes. Parameters are read-only and cannot appear on the left-hand side of assignments. Channels that are passed as parameters can be prefixed with either the `IN` or `OUT` modifiers to indicate that the process can only send or only receive on the channel. Channels without these keywords can be used for both sending and receiving.

An ESML model has the following structure:

```
MODEL name;
  CONST constant definitions
  TYPE type definitions
  VAR variable definitions

  process definitions

BEGIN
  activation commands
END name;

ASSERT correctness specification
```

Constant and type definitions are optional, but, when present, are global and visible to all processes. Only channel variables are allowed and this is the only place where they can have any function; local channel variables have no function.

The main body (between the last `BEGIN` and `END` keywords) may contain only activation commands and no activation commands are allowed in process bodies.

D.5 A grammar for ESML

Model

⟨*model*⟩ ::= `MODEL` name "`;`" ⟨*declarations*⟩ { ⟨*process*⟩ } ⟨*body*⟩ `ASSERT` ⟨*ctl formula*⟩.

⟨*process*⟩ ::= `PROCESS` name ⟨*parameter list*⟩ ⟨*declarations*⟩ ⟨*body*⟩.
⟨*body*⟩ ::= `BEGIN` ⟨*command list*⟩ `END` name "`;`".

Declarations

⟨*declarations*⟩ ::= [⟨*constant part*⟩] [⟨*type part*⟩] [⟨*variable part*⟩].
⟨*parameter list*⟩ ::= ["`(`" ⟨*parameter*⟩ { "`;`" ⟨*parameter*⟩ } "`)`"] "`;`".
⟨*parameter*⟩ ::= [`IN` | `OUT`] ⟨*variable definition*⟩.
⟨*constant part*⟩ ::= `CONST` ⟨*constant definition*⟩ { ⟨*constant definition*⟩ }.
⟨*constant definition*⟩ ::= name "`=`" ⟨*constant expression*⟩ "`;`".
⟨*type part*⟩ ::= `TYPE` ⟨*type definition*⟩ { ⟨*type definition*⟩ }.
⟨*type definition*⟩ ::= name "`=`" ⟨*type*⟩ "`;`".
⟨*type*⟩ ::= ⟨*subrange type*⟩ | ⟨*enum type*⟩ | ⟨*list type*⟩ | ⟨*array type*⟩
| ⟨*tuple type*⟩ | ⟨*alphabet type*⟩.
⟨*subrange type*⟩ ::= ⟨*constant expression*⟩ "`..`" ⟨*constant expression*⟩.
⟨*enum type*⟩ ::= name { "`,`" name }.
⟨*list type*⟩ ::= `LIST` "`[`" ⟨*constant expression*⟩ "`]`" `OF` name.
⟨*array type*⟩ ::= `ARRAY` "`[`" ⟨*constant expression*⟩ "`]`" `OF` name.
⟨*tuple type*⟩ ::= "`(`" ⟨*variable definition*⟩ { "`;`" ⟨*variable definition*⟩ } "`)`".
⟨*alphabet type*⟩ ::= "`{`" ⟨*symbol*⟩ { "`,`" ⟨*symbol*⟩ } "`}`".
⟨*symbol*⟩ ::= name ["`(`" name "`)`"].
⟨*variable part*⟩ ::= `VAR` ⟨*variable definition*⟩ "`;`"{ ⟨*variable definition*⟩ "`;`" }.
⟨*variable definition*⟩ ::= name { "`,`" name } "`:`" name.

Commands

⟨*command list*⟩ ::= ⟨*command*⟩ ["`;`" ⟨*command list*⟩].
⟨*command*⟩ ::= ⟨*access command*⟩ | ⟨*if command*⟩ | ⟨*do command*⟩ | ⟨*poll command*⟩
| ⟨*trace command*⟩ | ⟨*list command*⟩ | `SKIP`.
⟨*access command*⟩ ::= ⟨*variable access*⟩ ⟨*access*⟩.
⟨*access*⟩ ::= ⟨*assignment*⟩ | ⟨*io command*⟩ | ⟨*arguments*⟩.
⟨*assignment*⟩ ::= "`:=`" ⟨*expression*⟩.
⟨*io command*⟩ ::= ⟨*bang*⟩ | ⟨*hook*⟩.
⟨*arguments*⟩ ::= ["`(`" ⟨*expression*⟩ { "`,`" ⟨*expression*⟩ } "`)`"].
⟨*bang*⟩ ::= "`!`" name ["`(`" ⟨*expression*⟩ "`)`"].
⟨*hook*⟩ ::= "`?`" name ["`(`" ⟨*variable access*⟩ "`)`"].
⟨*if command*⟩ ::= `IF` ⟨*guard list*⟩ `END`.
⟨*do command*⟩ ::= `DO` ⟨*guard list*⟩ `END`.
⟨*guard list*⟩ ::= ⟨*guard*⟩ ["`[]`" ⟨*guard list*⟩].
⟨*guard*⟩ ::= ⟨*expression*⟩ "`->`" ⟨*command list*⟩.
⟨*poll command*⟩ ::= `POLL` ⟨*poll list*⟩ `END`.

⟨*poll list*⟩ ::= ⟨*poll*⟩ ["`[]`" ⟨*poll list*⟩].
⟨*poll*⟩ ::= ⟨*variable access*⟩ ⟨*io command*⟩ ["`&`" ⟨*expression*⟩]"`->`" ⟨*command list*⟩.
⟨*trace command*⟩ ::= `TRACE` "`(`" ⟨*trace expression*⟩ { "`,`" ⟨*trace expression*⟩ } "`)`".
⟨*trace expression*⟩ ::= ⟨*expression*⟩ | string.
⟨*list command*⟩ ::= `EMPTY` "`(`" ⟨*expression*⟩ "`)`" | `REMOVE` "`(`" ⟨*expression*⟩ "`)`"
| `APPEND` "`(`" ⟨*expression*⟩ "`,`" ⟨*expression*⟩ "`)`"
| `PREPEND` "`(`" ⟨*expression*⟩ "`,`" ⟨*expression*⟩ "`)`".

Expressions

⟨*constant expression*⟩ ::= ⟨*expression*⟩.
⟨*expression*⟩ ::= ⟨*primary*⟩ [⟨*primary operator*⟩ ⟨*expression*⟩].
⟨*primary operator*⟩ ::= "`&`" | `OR`.
⟨*primary*⟩ ::= ⟨*secondary*⟩ [⟨*secondary operator*⟩ ⟨*primary*⟩].
⟨*secondary operator*⟩ ::= "`<`" | "`<=`" | "`>`" | "`>=`" | "`=`" | "`#`".
⟨*secondary*⟩ ::= ⟨*term*⟩ [⟨*adding operator*⟩ ⟨*secondary*⟩].
⟨*adding operator*⟩ ::= "`+`" | "`-`".
⟨*term*⟩ ::= ⟨*factor*⟩ [⟨*multiplying operator*⟩ ⟨*term*⟩].
⟨*multiplying operator*⟩ ::= "`*`" | `DIV` | `MOD`.
⟨*factor*⟩ ::= number | `TRUE` | `FALSE` | ⟨*variable access*⟩ | ⟨*list operation*⟩
| "`(`" ⟨*expression*⟩ "`)`" | "`-`" ⟨*expression*⟩ | "`~`" ⟨*factor*⟩.
⟨*list operation*⟩ ::= `LEN` "`(`" ⟨*expression*⟩ "`)`" | `HEAD` "`(`" ⟨*expression*⟩ "`)`".
⟨*variable access*⟩ ::= name { "`.`" name | "`[`" ⟨*expression*⟩ "`]`" }.

CTL

⟨*ctl formula*⟩ ::= ⟨*subformula*⟩ [⟨*ctl operator*⟩ ⟨*ctl formula*⟩].
⟨*ctl operator*⟩ ::= "`&`" | `OR` | "`=>`".
⟨*subformula*⟩ ::= `AG` ⟨*ctl formula*⟩ | `EG` ⟨*ctl formula*⟩ | `AF` ⟨*ctl formula*⟩ | `EF` ⟨*ctl formula*⟩
| `AX` ⟨*ctl formula*⟩ | `EX` ⟨*ctl formula*⟩ | `A` "`(`" ⟨*ctl formula*⟩ `U` ⟨*ctl formula*⟩ "`)`"
| `E` "`(`" ⟨*ctl formula*⟩ `U` ⟨*ctl formula*⟩ "`)`" | "`~`" ⟨*subformula*⟩ | "`(`" ⟨*ctl formula*⟩ "`)`"
| ⟨*expression*⟩.

Tokens

name ::= letter { letter | digit }.
number ::= digit { digit }.
letter ::= "`a`" | ... | "`z`" | "`A`" | ... | "`Z`".
digit ::= "`0`" | ... | "`9`".
string ::= "`"`" { char } "`"`".
char ::= Any printable ASCII character.

Comments

Comments are delimited by "`(*`" and "`*)`" and may be nested.

List of keywords

A	AF	AG	APPEND	ARRAY
ASSERT	AX	BEGIN	CONST	DIV
DO	E	EF	EG	EMPTY
END	EX	FALSE	HEAD	IF
IN	LEN	LIST	MOD	MODEL
OF	OR	OUT	POLL	PREPEND
PROCESS	REMOVE	SKIP	TRACE	TRUE
TYPE	U	VAR		

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] D. C. Barnard. Reducing the state explosion problem during model checking. Master's thesis, University of Stellenbosch, March 1991.

[3] H. Barringer, M. Fischer, and G. Gough. Fair SMG and linear time model checking. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science #407, pages 133–150. Springer-Verlag, June 1989.

[4] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *CAV'94: Proceedings of the 6th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science #818, pages 142–155. Springer-Verlag, June 1994.

[5] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, pages 388–397. IEEE Computer Society Press, June 1995.

[6] P. Brinch Hansen. *Programming a Personal Computer*. Prentice Hall, Englewood Cliffs, 1982.

[7] P. Brinch Hansen. Joyce—a programming language for distributed systems. *Software—Practice and Experience*, 17(1):29–50, January 1987.

[8] P. Brinch Hansen. A Joyce implementation. *Software—Practice and Experience*, 17(4):267–276, April 1987.

[9] D. R. Cheriton. VMTP: a transport protocol for the next generation of communication systems. In *Proceedings of the 4th Symposium on Communications Architectures and Protocols*, pages 406–415. ACM, August 1986.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic in Programs: Workshop, Yorktown Heights, NY*, Lecture Notes in Computer Science #131, pages 52–71. Springer-Verlag, May 1981.

[11] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[12] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *CAV'90: Proceedings of the 2nd International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science #531, pages 233–242. Springer-Verlag, June 1990.

[13] P. J. A. de Villiers. A model checker for transition systems. In *Proceedings of the 6th Southern African Computer Symposium*, pages 262–275, July 1991.

[14] P. J. A. de Villiers and W. C. Visser. ESML—a validation language for concurrent systems. In *Proceedings of the 7th Southern African Computer Symposium*, 1992.

[15] C. Demartini, R. Iosif, and R. Sisto. dSPIN: a dynamic extension of SPIN. In *Proceedings of the 6th SPIN Workshop*, September 1999.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

[17] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier/The MIT Press, 1990.

[18] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching time versus linear time. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 127–140, January 1983.

[19] E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, January 1987.

[20] J.-C. Fernandez and L. Mounier. Verifying bisimulation on-the-fly. In *FORTE'90: Proceedings of the IFIP TC6/WG6.1 3rd International Conference on Formal Description Techniques*, 1990.

[21] N. Francez. *Fairness*. Springer-Verlag, 1986.

[22] J. Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *Proceedings of the 5th SPIN Workshop*, Lecture Notes in Computer Science #1680, pages 12–21. Springer-Verlag, July 1999.

[23] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. In *Proceedings of the 3rd Israeli Symposium on the Theory of Computing and Systems*, pages 130–139, 1995.

[24] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV'90: Proceedings of the 2nd International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science #531, pages 176–185. Springer-Verlag, June 1990.

[25] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proceedings of the 13th IFIP Symposium on Protocol Specification, Testing, and Verification*, pages 109–124, May 1993.

[26] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *CAV'92: Proceedings of the 4th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science #663, pages 175–186. Springer-Verlag, June 1992.

[27] J.-C. Grégoire. State space compression in SPIN with GETSs. In *Proceedings of the 2nd SPIN Workshop*, August 1996.

[28] D. Gries. Is sometime ever better than alway? *ACM Transactions on Programming Languages and Systems*, 1(2):258–265, October 1979.

[29] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[30] G. J. Holzmann. *Basic SPIN Manual.* AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[31] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(10):2413–2433, December 1985.

[32] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proceedings of the 6th IFIP Symposium on Protocol Specification, Testing, and Verification*, June 1987.

[33] G. J. Holzmann. An improved protocol reachability analysis technique. *Software—Practice and Experience*, 18(2):137–161, February 1988.

[34] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series, 1991.

[35] G. J. Holzmann. State compression in SPIN: recursive indexing and compression training runs. In *Proceedings of the 3rd SPIN Workshop*, April 1997.

[36] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, November 1998.

[37] G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 3(1), 1999.

[38] P. Klint. Interpretation techniques. *Software—Practice and Experience*, 11(9):963–973, November 1981.

[39] L. Lamport. "Sometime" is sometimes "not never"—on the temporal logic of programs. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[40] H. L. Loedolff, P. J. A. de Villiers, and W. C. Visser. Validation of reactive programs which incorporate structured data. Technical Report RW/VER/93/01/01, Department of Computer Science, University of Stellenbosch, August 1993.

[41] Z. Manna and A. Pnueli. Verification of concurrent programs, part I: the temporal framework. Technical Report STAN–CS–81–836, Department of Computer Science, Stanford University, July 1981.

[42] Z. Manna and R. Waldinger. Is "sometime" sometimes better than "always"? *Communications of the ACM*, 21(2):159–172, February 1978.

[43] K. L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, May 1992.

[44] D. Peled. All from one, one for all: on model checking using representatives. In *CAV'93: Proceedings of the 5th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science #697, pages 409–423. Springer-Verlag, June 1993.

[45] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the 18th IEEE Symposium on the Foundation of Computer Science*, pages 46–57. IEEE Computer Society Press, October 1977.

[46] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency: Overview and Tutorials*, Lecture Notes in Computer Science #224, pages 510–584. Springer-Verlag, 1986.

[47] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium in Programming*, Lecture Notes in Computer Science #137, pages 337–351. Springer-Verlag, 1982.

[48] J. Rushby. Mechanized formal methods: Progress and prospects. In *Proceedings of the 16th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science #1180, pages 43–51. Springer-Verlag, December 1996.

[49] A. S. Tanenbaum. Implications of structured programming for machine architecture. *Communications of the ACM*, 21(3):237–246, March 1978.

[50] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[51] A. Valmari. Error detection by reduced reachability graph generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988.

[52] A. Valmari. A stubborn attack on state explosion. In *CAV'90: Proceedings of the 2nd International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science #531, pages 156–165. Springer-Verlag, June 1990.

[53] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science #1491, pages 429–528. Springer-Verlag, 1998.

[54] J. L. A. van de Snepscheut. The sliding-window protocol revisited. *Formal Aspects of Computing*, 7(1):3–17, 1995.

[55] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, June 1986.

[56] B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In *CONCUR'93: Proceedings of the 4th International Conference on Concurrency Theories*, Lecture Notes in Computer Science #715, pages 447–461. Springer-Verlag, August 1993.

[57] W. C. Visser. An execution environment for a validation language. In *Proceedings of the 7th National Masters and PhD Computer Science Students Conference*, pages 233–244, July 1992.

[58] W. C. Visser. A run-time environment for a validation language. Master's thesis, University of Stellenbosch, October 1993.

[59] W. C. Visser. Memory efficient state storage in SPIN. In *Proceedings of the 2nd SPIN Workshop*, pages 21–35, August 1996.

[60] D. H. D. Warren. An abstract PROLOG instruction set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, October 1983.

[61] N. Wirth. Pascal-S: a subset and its implementation. Technical report, ETH Zürich, June 1975.

[62] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley Publishing Company, 1992.

[63] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *CAV'93: Proceedings of the 5th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science #697, pages 59–70. Springer-Verlag, June 1993.

Bibliographic crossreference

[1] 68, 70, 71

[2] 47

[3] 11

[4] 11

[5] 11

[6] 24

[7] 120

[8] 120

[9] 95

[10] 7, 8, 9

[11] 98

[12] 11

[13] 47

[14] 20, 29, 47, 120

[15] 33

[16] 7, 22, 125

[17] 8

[18] 7, 14

[19] 7, 14

[20] 11

[21] 14

[22] 41, 43, 80

[23] 12, 98

[24] 12

[25] 86, 94

[26] 36, 76, 81, 83

[27] 36

[28] 7

[29] 120

[30] 28, 94, 120

[31] 34

[32] 35

[33] 35

[34] 2, 21, 35, 61, 94, 120

[35] 40, 41

[36] 35

[37] 36

[38] 24

[39] 7

[40] 47

[41] 7

[42] 7

[43] 17

[44] 12

[45] 7, 61

[46] 92

[47] 9

[48] 98

[49] 24

[50] 45, 68

[51] 12

[52] 12

[53] 12

[54] 77

[55] 11

[56] 11

[57] 95

[58] 21, 41, 47, 92

[59] 36, 40

[60] 24

[61] 24

[62] 47

[63] 35