A Run-time Environment for a Validation Language

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF THE UNIVERSITY OF STELLENBOSCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Willem Conradie Visser
11 October 1993

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: ....

Abstract

Our Department is currently engaged in a project to validate the correctness of reactive systems, specifically operating system kernels. Model checking is used as a validation technique. A model checker was implemented using transition systems as a modelling formalism and computation tree logic (CTL) to specify correctness requirements. Although transition systems are powerful enough to specify the behaviour of reactive systems, it is inconvenient to use because it is too low level. Therefore a high-level validation language is required. Since the behaviour of an operating system kernel is often dependent on the manipulation of complex data the validation language must support complex data structures.

This thesis describes the design and implementation of a compiler and run-time environment for a high-level validation language. The validation language ESML supporting records and lists is used for the efficient modelling of reactive systems. The compiler translates an ESML model into an equivalent transition system which is used as input by the model checker. During model checking the transitions are executed in an efficient run-time environment. A compaction technique is implemented during run-time that allows memory efficient model checking of ESML models with complex data. The design and implementation of the ESML compiler and run-time environment are described and compared to existing systems.

Model checking has a serious limitation: the model checker has to generate the reachable state space of a model and this state space can be very large. To counter this *state explosion* problem, three model reduction techniques are described. Efficient implementations of these techniques for the reduction of an ESML model's state space are given. A summary of the performance of the reduction techniques is provided.

Acknowledgements

I am indebted to all the people who have assisted me throughout my studies, especially

- Pieter de Villiers and Prof. A.E. Krzesinski;
- Dieter Barnard, Hans Loedolff, Pieter Muller, Abriëtte Senekal and Lynette Lewis;
- My parents, family and friends;
- The Department of Computer Science;
- Infoplan;
- The Foundation for Research and Development (FRD).

I would like to express my sincere gratitude to all.

Contents

Abstract **iii**

Acknowledgements **iv**

1 Introduction **1**

1.1 Thesis Goal 3

1.2 Thesis Layout 3

2 Model Checking **5**

2.1 Temporal Logic 6

2.1.1 CTL 7

2.1.2 Specifications 8

2.1.3 An Example 9

2.2 Transition Systems as Computational Model 10

2.2.1 An Example of a Transition System 11

2.3 Model Checkers 12

2.3.1 "On-the-fly" Model Checking 13

2.3.2 Example: Model Checking for Mutual Exclusion 14

2.4 Summary 16

3 **A Validation Language for Reactive Systems** **17**

3.1 ESML 18

3.1.1 Type and Variable Definitions 19

3.1.2 Expressions 24

3.1.3 Control Structures 24

3.1.4 ESMs : Definition and Activation 27

3.1.5 Scope Rules 29

3.1.6 Requirement Specifications 29

3.2 ESML Examples 30

3.3 A Validation System 33

4 **The Transition System** **36**

4.1 The Transition Format 37

4.1.1 Fundamental Instructions 38

4.1.2 Guarded Commands 39

4.2 CTL formulae 42

4.3 The Modula-2 Transition System 43

5 **A Run-Time Environment for ESML** **47**

5.1 Storage Organisation 49

5.1.1 Communication Channels 49

5.1.2 Activation Records 51

5.1.3 Compact Variable Representation 54

5.1.4 State Vector Access 58

5.2 State Generation 61

5.2.1 ESM Scheduling 61

5.2.2 Generating All Execution Paths 62

5.2.3 The NextState Procedure 63

5.3 CTL Formula Evaluation 67

6 The ESML Compiler **68**

6.1 Parser 68

6.1.1 Symbol Table 69

6.1.2 Abstract Syntax Graph 70

6.2 Code Generation 72

6.3 Results 76

7 Optimisations **79**

7.1 Model Reduction 79

7.1.1 Sleep Sets 80

7.1.2 Partial Order Semantic Rules 88

7.1.3 Transition Folding 90

7.1.4 Results 91

7.2 Future Work 92

8 Conclusions **93**

8.1 Retrospective 93

8.2 Program Correctness 95

A Scheduler Model **97**

List of Tables

1 Validation results 77

2 Results obtained by adding model reduction techniques. 92

List of Figures

1 A computation tree for each CTL operator 8

2 Reachability graph for the mutual exclusion system 10

3 A state vector for the mutual exclusion system 15

4 Reachability graph labelled with state values and transition numbers. . . . 15

5 An ESML model of a recursive factorial algorithm 31

6 An ESML model of the alternating bit protocol 32

7 The proposed validation system 33

8 The structure of the model checker and transition system 48

9 The state vector and activation list for the mutual exclusion model 53

10 A symbol table (right) after the compilation of the ESML code (left) . . . 70

11 An abstract syntax graph for an IF construct. 71

12 Syntax graph for `ESM Code`. 73

13 An ESML model 80

14 The reachability graph traversed during a depth-first search 81

15 A depth-first search algorithm using sleep sets 83

16 The reachability graph traversed during a depth-first search with sleep sets 84

17 A depth-first search algorithm for ESML models using sleep sets. 86

18 The reachability graph traversed guided by the partial order rule. 89

19 The reachability graph traversed with transition folding added. 91

Chapter 1

Introduction

Computer systems are being used in safety critical applications such as in aircraft, medical equipment and nuclear power stations. These systems are often concurrent in nature and can be complex. Until now software design methodologies and rigorous testing procedures were the accepted methods to establish system correctness. Unfortunately, the necessary level of correctness cannot always be obtained by these methods [15, 38]. However, the use of formal methods to prove system correctness are gaining ground as a promising alternative to these techniques.

Formal methods require a system specification to be given. There are two approaches for doing this: *single* language and *dual* language [74]. In the single language approach only the *behaviour* of a system is specified. The formality of this approach forces the designer to think carefully about the intended behaviour and thus helps to eliminate errors. Current specification languages adopting the single language approach include VDM [59] and Z [82]. Manna and Pnueli introduced the dual language approach: a set of *correctness requirements* are specified in addition to specifying the behaviour of a system [66, 68, 75]. The advantage of the dual language approach is that a *validation tool* can be used to check the behavioural specification against the set of requirements.

During the past decade *model checking* [19, 20] has been established as a successful validation technique for concurrent systems [11, 24, 63]. Model checking establishes the correctness of a system by checking that the behavioural specification of a system conforms to its correctness requirements. A *state-transition graph* is used to represent the behaviour of a

concurrent system. Each state consists of a set of variables describing the execution state of each process in the concurrent system. *Temporal logic* is suitable for specifying many correctness requirements of concurrent systems. Propositional logic can only express properties that hold for the present whereas temporal logic can express changes over time. For instance a typical temporal formula can express "property p will eventually hold". A model checker therefore establishes whether a temporal logic formula holds in the state-transition graph of a system.

A locally developed model checker [26] uses a transition system as behavioural specification and generates the state-transition graph on-the-fly during the model checking procedure. On-the-fly graph generation is an improvement on generating the complete state-transition graph beforehand, because depending on the temporal formula to be validated only part of the graph may be generated. Although transition systems are powerful enough to specify the behaviour of concurrent systems, it is inconvenient to use directly because it is too low level. A high-level validation language is required to specify the behaviour of a concurrent system.

Communication protocols, operating system kernels, embedded systems and process control systems are examples of a special class of concurrent systems called reactive systems [47]. An important property of reactive systems is that they interact continuously with their environments and do not compute a final value on termination—in fact, they are usually designed not to terminate at all. The modelling and validation of reactive systems have enjoyed widespread interest [47, 69, 76]; most work having been done in the area of communication protocols. We are interested in the validation of operating system kernels which has the interesting property that their behaviour often depends on the manipulation of complex data.

The high-level validation language ESML (Extended State Machine Language) was designed to model reactive systems. To facilitate the modelling and validation of kernels ESML supports the complex data structures records and lists. Unfortunately, currently available validation systems [24, 44, 51, 61, 69, 71] can analyse validation models with complex control structures but only *simple data structures*. A validation system is therefore required that can validate the correctness of ESML models.

1.1 Thesis Goal

A compiler is required to generate an equivalent transition system from a high-level behavioural specification. This thesis describes the design and implementation of a compiler and run-time environment for a high-level validation language incorporating complex data structures. The compiler translates an ESML model into an equivalent transition system which is used as input by the model checker. During model checking the transitions are executed in an efficient run-time environment. The run-time environment of the protocol validation language *Promela* was used as a guideline during development [51]. A compaction technique is described whereby any variable in an ESML model will always be stored in the *minimum* amount of memory required. This compaction technique allows memory efficient model checking of ESML models with complex data. To counter the *state explosion* problem during model checking [4] three model reduction techniques are described. These techniques reduce the memory requirements for the validation of ESML models by reducing the number of states stored during model checking.

1.2 Thesis Layout

The rest of the thesis is structured as follows:

Chapters 2 and 3 supply the necessary background for the remainder of the thesis. **Chapter 2** provides an introduction to model checking and specifically a model checker for transition systems. A brief overview of the first model checking algorithm, published by Clarke and Emerson [19], is given. The branching time temporal logic *CTL* (Computation Tree Logic) used to specify correctness requirements is described. The chapter concludes by describing an efficient model checker for transition systems together with an example. The high-level validation language ESML is introduced in **Chapter 3**. The chapter includes a discussion of the initial design decisions as well as a description of the syntax and semantics of the language. The chapter concludes with the design of a proposed validation system incorporating both a simulator and model checker for ESML models.

Chapters 4 through 7 are concerned with the model checking of ESML models. Firstly, in **Chapter 4**, the transition system to be used as input by the model checker is defined.

Chapter 5 describes the run-time environment for executing the transitions during model checking. The first part of the chapter is devoted to the storage management during the execution of transitions. Comparisons with the implementation of the run-time environment of the concurrent programming language Joyce [9, 10] and the protocol validation language Promela [51] are given. A compaction technique reducing the memory requirements of the model checker is described. In the second part of the chapter state generation and CTL formula evaluation during model checking are described. In **Chapter 6** the ESML compiler which performs the translation of ESML models into transition systems is described. The chapter concludes by giving results obtained from compiling and model checking a number of ESML models. These models include a process scheduler, an elevator and a part of the X-Windows system. In **Chapter 7** three optimisation techniques are described for reducing the time and memory requirements for the model checking of ESML models. A summary of the performance of the reduction techniques is provided.

The final chapter presents a retrospective on the validation system as well as a brief discussion of program correctness in general.

Chapter 2

Model Checking

The verification of concurrent systems is complex due to the intricate interaction of different processes. Although theorem proving has been used to verify a number of hardware systems [25], user interaction is required whenever part of a proof cannot be resolved. Expert knowledge is therefore needed to verify a system by using a theorem prover. Model checking [19, 20] is a powerful alternative to theorem proving. The model checking technique can be fully automated and thus needs no user assistance.

Systems of practical size are in general too complex to verify directly. However, by ignoring irrelevant detail, a model which represents the intended behaviour of a concurrent system can be designed. For this, a validation language is required and the resulting model is called the *behavioural specification*. In addition, temporal logic can be used to specify many important correctness properties of concurrent systems. This is called the *requirement specification*. A model checker is used to determine whether the behavioural specification conforms to the requirement specification. Execution of a concurrent system is modelled as a sequence of state transitions, starting from a specified initial state. All variables accessed and modified by processes in the system collectively describe the state of the system. Model checking has been used successfully to verify systems such as hardware modules [11], protocols [24, 63] and even real time systems [79].

On the negative side a serious limitation of model checking is the potentially huge number of states generated by complex models. Fortunately this so-called *state explosion* problem can be countered by various techniques [4, 49, 41] which are designed to reduce the number of

states generated. An overview of a locally developed model checker [26] which incorporates several such state space reduction techniques is given in section 2.3.1.

2.1 Temporal Logic

When dealing with concurrent systems more than the input-output behaviour of a system is important. In fact, the entire execution sequence must be considered. *Temporal logic* which can express changes over time is suitable for specifying many correctness properties of concurrent systems. The power of temporal logic as a specification language has been demonstrated in various application areas such as concurrent programs [64, 65, 85], protocols [28, 35], hardware [7, 45] and real-time systems [74].

Manna and Pnueli introduced a *linear time* temporal logic that is useful to describe execution sequences of concurrent systems [66]. The logic extends classical propositional logic by adding four non-truthfunctional temporal operators: *always* ($\Box$), *sometimes* ($\Diamond$), *next* ($\bigcirc$) and *until* ($\mathcal{U}$). Three classes of properties of concurrent systems were identified, namely *safety*, *liveness* and *precedence* properties [68, 76].

Every execution of a program yields a computation which is a sequence of states. These computations could be assembled into a structure that represents the behaviour of a program in more than one way. One approach considers the behaviour of a program to be the *set* of its computations. The appropriate temporal logic to describe such behaviours is *linear time* temporal logic. Alternatively, the individual computations are assumed to form a *computation tree*. *Branching time* temporal logic defines modalities over trees of states and is consequently the appropriate logic for this approach.

The choice between linear and branching time temporal logic caused some controversy [18, 32, 33, 76]. Linear time logic expresses properties that must hold for *all* execution paths of a program— it cannot express properties that hold for *some* execution paths of a program. Branching time logic on the other hand can express properties holding along some paths as well as all paths in the computation tree. However, Manna and Pnueli's linear time logic can express fairness [66, 67] whereas the branching time logic CTL (*Computation Tree Logic*) cannot [12, 20, 32]. CTL, however, allows efficient model checking [19]. For

this reason several validation systems are based on CTL [12, 31].

2.1.1 CTL

Clarke and Emerson first used CTL to specify and model check concurrent systems [19]. The base of CTL, as defined in [19], is the propositional calculus on $\wedge$ ("and") and $\neg$ ("not") with the other operators $\vee$ ("or") and $\Rightarrow$ ("implies") defined in the usual way. Modal operators consist of two symbols: a path quantifier, either A ("for all computation paths") or E ("there exists a computation path"), followed by one of the state quantifiers G ("always"), F ("sometimes"), X ("next") or U ("until"). The formal syntax of CTL formulae can be found in [21].

The semantics of CTL is defined with respect to a Kripke structure $K = (S, R, v)$ where

1. S is a finite set of states. Some unique state $s_0 \in S$ is defined as the *initial state*.
2. R is a binary relation on S which defines the possible transitions between states ($R \subseteq S \times S$).
3. v is an assignment of truth values to every proposition at every state ($v : P \times S \rightarrow$ { True, False } where P is the set of propositions of a given CTL formula).

A path $\pi = s_0, s_1, s_2, \ldots$ is a sequence of states, leading from the root state s_0 in a computation tree, such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. A formal definition of the CTL operators are given in [21]. Here an informal description will suffice:

$AG(f)$: Formula f holds at *every* state on *every* path from s_0 (the initial state).

$EG(f)$: Formula f holds at *every* state on *some* path from s_0.

$AF(f)$: Formula f holds in *some* state along *every* path from s_0.

$EF(f)$: Formula f holds in *some* state along *some* path from s_0.

$AX(f)$: Formula f holds at *every* immediate successor state of s_0.

$EX(f)$: Formula f holds at *some* immediate successor state of s_0.

$A(f_1 U f_2)$: Formula f_1 holds at *every* state on *every* path from s_0 *until* formula f_2 holds (f_1 and f_2 can be true in the same state, but need not be).

$E(f_1 U f_2)$: Formula f_1 holds at *every* state along *some* path from s_0 *until* formula f_2 holds (f_1 and f_2 can be true in the same state, but need not be).

A computation tree satisfying each CTL operator is given in Figure 1.

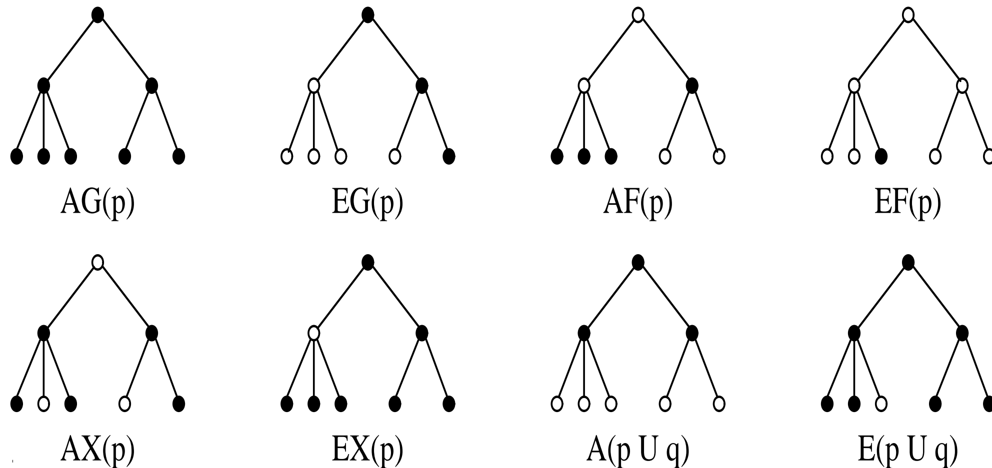


Figure 1: A computation tree for each CTL operator. *Let* $\bullet = p$ *and* $\circ = \neg p \wedge q$.

2.1.2 Specifications

Ensuring the *completeness* of the requirements specification of a system is important. If the requirements are incomplete a system can be proven correct according to its specification, although it can still behave incorrectly in certain situations. Currently ensuring completeness is left to the user ("specifier"); no formal method is available for generating the requirements for a concurrent system. However, the three classes of properties mentioned in section 2.1 provide a guideline for determining the requirement specifications of a concurrent system.

Safety Properties: state that "something bad never happens" (a program never enters an unacceptable state).

Liveness Properties: state that "something good will eventually happen" (a program eventually enters a desirable state).

Precedence Properties: states that "nothing bad will happen" until "something good happens" (a program will not enter an unacceptable state before it enters a desirable state).

These properties can be expressed in linear time temporal logic as well as CTL. In CTL safety properties can be expressed by formulae of the general form $AG(f)$, liveness properties by $AF(f)$ and precedence properties by $A(f_1 U f_2)$.

2.1.3 An Example

Consider the mutual exclusion problem for two processes where each process ($i = 1, 2$) can be in one of three code regions: noncritical (N_i), trying (T_i) or critical (C_i). A binary semaphore S is used to protect the critical region. The value of the semaphore is indicated by S_i in Figure 2, where i can be 0 or 1. A process can only enter its *critical* region from its *trying* region if the value of the semaphore is 0. When a process enters its critical region the value of the semaphore becomes 1 and on leaving the critical region and entering the *noncritical* region the value of the semaphore becomes 0 again.

The *reachability graph* of a system is a labelled state transition graph of all reachable states. Each state is labelled with the values of all variables in that state. The computation tree of a system can be produced by unfolding its reachability graph. The reachability graph of the mutual exclusion system described above is shown in Figure 2. Each node is labelled with the code region currently executed by each process and the state of the semaphore.

Here are three examples of requirement specifications:

Safety Property (Mutual Exclusion): The two processes will never be in their critical regions at the same time. $AG(\neg(C_1 \wedge C_2))$

Liveness Property (Absence of Starvation): Once a process has entered its trying region it will eventually enter its critical region. $AG(T_i \rightarrow AF(C_i))$ with ($i = 1, 2$)

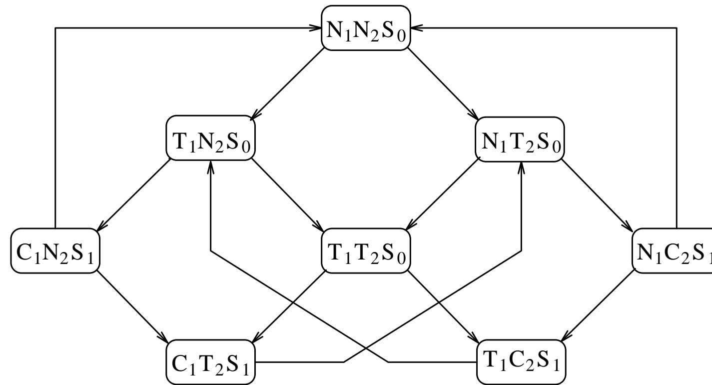


Figure 2: Reachability graph for the mutual exclusion system

Precedence Property (Safe Liveness): Neither of the two processes can enter their critical regions until one of them receives the semaphore (semaphore becomes 1). $A(\neg(C_1 \vee C_2)\ U\ S_1)$

2.2 Transition Systems as Computational Model

Early model checking systems stored the complete reachability graph in memory. The main disadvantage of this approach is that for a large concurrent system the number of states becomes unmanageable. This prompted a new direction in model checking called partial graph generation [52, 57, 58]. Only those parts of the reachability graph needed for model checking a specific CTL formula are computed "on-the-fly". This technique is used in the model checker discussed in section 2.3.1.

A transition system can be used to generate the reachability graph of a system and can thus serve as input format for a model checker. A transition system describing the behaviour of processes in a concurrent system can be defined as a triple $\mathcal{TS} = (S, T, s_0)$ where

- S is a non-empty set of states. A state consists of the set of variables accessed and modified by the processes in a concurrent system. Each process contributes a control variable (to indicate the current point of execution in the process) and a possibly empty set of data variables. Because each variable has a range of values associated with it, the state can be represented by a vector of bits called the *state vector*. The state vector for the mutual exclusion system can be seen in Figure 3.

- T is a non-empty set of transitions. A transition is a tuple, $\tau = (g_\tau, a_\tau)$, where g_τ is an *enabling condition* (*guard*) for its corresponding *action* a_τ. The guard is *satisfied* in a state s if it is true in s. A transition τ is *enabled* in state s if its guard g_τ is *satisfied* in s. The action a_τ is a partial function (defined on states $s \in S$ in which τ is enabled) given by $a_\tau : S \to S$. If a transition is enabled in a state s, it can be executed to generate a new state s' from s with $s' = a_\tau(s)$.

- s_0 is the start state of the transition system with $s_0 \in S$.

2.2.1 An Example of a Transition System

Consider the mutual exclusion problem described in section 2.1.2. The state in the transition system consists of the values of the control variable for $process_1$, $process_2$ and the $semaphore$ variable and is described by $(process_1, process_2, semaphore)$. An asterisk ("*") in the guard of a transition indicates that the field can hold any value for the guard to be satisfied in that state. The action part of the transition will indicate how each field is changed, with "*" indicating unchanged fields. The start state is (N_1, N_2, S_0) and the transitions are the following:

$$
\begin{aligned}
(N_1, *, *) &\rightarrow (T_1, *, *) && (1)\\
(T_1, *, S_0) &\rightarrow (C_1, *, S_1) && (2)\\
(C_1, *, *) &\rightarrow (N_1, *, S_0) && (3)\\
(*, N_2, *) &\rightarrow (*, T_2, *) && (4)\\
(*, T_2, S_0) &\rightarrow (*, C_2, S_1) && (5)\\
(*, C_2, *) &\rightarrow (*, N_2, S_0) && (6)
\end{aligned}
$$

Successor states of a state s are generated by executing all enabled transitions. For example in the initial state of the mutual exclusion transition system above both transitions 1 and 4 are enabled. Executing transition 1 will generate the successor state (T_1, N_2, S_0) and executing transition 4 will result in state (N_1, T_2, S_0) being generated from the initial state. The reachability graph given in Figure 2 can be constructed by executing the transitions given above and labelling each state in the graph with the values of the state fields.

The complete reachability graph for a finite state system modelled by a transition system can be computed in the following way:

1. Initialise the graph with the root being the start state s_0. Initialise the *working set* of states W with s_0. W will always contain the states for which successor states must still be generated.
2. Take a state s from W and execute the transitions in the transition system to generate *all* the *successor* states s' of s. If the successor states are not already in the graph, add them to W and add them to the graph by inserting an arc from s to each state s'. If s has no successor states or all its successor states are already in the graph, remove s from W.
3. Repeat step 2 until W is empty.

This algorithm will terminate because s is removed from the working set W when all its successor states have been generated, only new states are added to W and it is assumed to be a finite state system.

2.3 Model Checkers

A combination of transition systems and temporal logic is now possible. A concurrent system is represented by a transition system from which a reachability graph can be computed. The graph consists of a finite set of states and arcs between the states which define a binary relation on the states. By testing the values of the variables in each state of the graph the truth value of atomic propositions can be determined. The reachability graph generated from the transition system can thus be interpreted as a Kripke structure. CTL formulae express desired properties that must hold for a given transition system and a model checker is used to determine whether such CTL formulae are satisfied by the reachability graph.

Clarke and Emerson published the first model checking algorithm [19]. A reachability graph is generated and stored in memory before the model checking algorithm starts. The algorithm proceeds bottom-up by calculating the truth values for each subformula of a CTL formula in every state of the labelled transition graph, starting with the shortest subformulae. The simplest possible subformulae are propositions, and their truth values are already known from the labels in each state. The truth values of the longer subformulae are determined from the semantics of CTL. Later Clarke, Emerson and Sistla improved the model checking algorithm to make it faster [20, 21]. The amount of available memory determines the size of the reachability graph that can be stored and therefore limits the size of the problems that can be handled.

Symbolic model checking, whereby the state space is represented symbolically rather than explicitly has been proposed in [14, 70]. *Binary decision diagrams* (BDDs) [13] are used to represent both the transition system and the state space generated during model checking. Although BDDs are very efficient when used to determine the truth value of a formula, they can be large and difficult to construct in certain cases. For instance the BDD representation of a multiplication operation can become too large to handle [17].

2.3.1 "On-the-fly" Model Checking

The work presented here forms part of an ongoing project at the University of Stellenbosch. The goal of the project is to develop a model checking system capable of validating a microkernel—the foundation of most modern operating systems. This model checker, which incorporates several effective techniques to counter the state explosion problem, will be described briefly. For a detailed description the reader is referred to [26].

It is unnecessary to generate the complete reachability graph before model checking—it is better to construct the reachability graph"on-the-fly" by only generating new states as needed to evaluate a given CTL formula. Furthermore, generating the reachability graph in a depth-first manner ensures that only the current execution path, namely the path from the initial state to the current state, need to be stored. This approach has three important advantages:

- Space is saved because the complete reachability graph is not stored explicitly.
- Time is saved due to the fact that the truth value of many CTL formulae can be determined without generating the entire reachability graph. An example of such a formula is $EF\alpha$ which will be satisfied as soon as a state is found in which α is true.
- Models which generate a reachability graph that is too large to fit into memory can sometimes be analysed because it may be unnecessary to generate the entire graph.

The success of model checking (or state space exploration in general) depends on efficient state comparison techniques. Each new state generated must be checked against all previous states to prevent the model checker from evaluating a state more than once. Holzmann

introduced an efficient method of determining uniqueness of states [49, 50, 51]. This technique requires a large vector of bits (of fixed size) to be maintained in memory to keep track of previously generated states. A hashing technique is used to compute an index into this bit vector from the value of each state. However, if a hash conflict occurs the validation results obtained can be invalid. To avoid this problem it is useful to note that the bit vector is extremely sparse and clustered [49]. This allows paging techniques to be used for the allocation of the memory needed for the bit vector. These pages are allocated in main memory whenever a bit is indexed which falls outside any of the already allocated pages. No secondary storage is used in this technique—that has been shown to be too slow [49]. The paging technique was implemented and found to work well for highly clustered state spaces [26]. Godefroid *et al.* later sketched and analysed a similar paging technique which they call *Hybrid* storage [40].

The current execution path is stored in a stack format. When a new state is generated via the transition system it is pushed onto the stack and becomes the current state. A stack entry at the top-of-stack can be removed (popped) in two possible situations: firstly, when the truth value of the CTL formula currently being evaluated has been determined and secondly, if no transition is enabled in the current state. When a stack record is removed, the state at the new top-of-stack becomes the current state.

The model checker only considers *fair* execution paths. Nested CTL formulae are handled in a top-down fashion to ensure that truth values are only computed when necessary. Fairness [37] and the handling of nested CTL formulae fall beyond the scope of this thesis. The reader is referred to [26] for a detailed discussion of these topics in the model checker.

2.3.2 Example: Model Checking for Mutual Exclusion

Consider the mutual exclusion example of section 2.1.2. The two control variables each has three possible values: N_i (0), T_i (1) or C_i (2). The semaphore has two values: S_0 (0) or S_1 (1). The two control variables will thus have two bits and the semaphore one bit allocated for it in the state vector (Figure 3).

When using this enumeration of values the cardinal value of each state can be easily determined. For instance the state (T_1, T_2, S_0) means that $process_1$ and $process_2$ has value

State vector: 5 bits		
$Process_1$	$Process_2$	Semaphore
4..3	2..1	0

Figure 3: A state vector for the mutual exclusion system

1 and the *semaphore* value 0. Therefore bits 1 and 3 will be set in the state vector resulting in the vector of bits $(0,1,0,1,0)$ (10 when interpreted as a cardinal value). Similarly state (C_1, T_2, S_1) results in the state vector $(1,0,0,1,1)$ which translates to value 19. The bit vector for this state vector with 5 bits will have $2^5 = 32$ entries. In Figure 4 the reachability graph of Figure 2 is changed to indicate the value of each state and the transition (section 2.2) that is executed to generate the following state. Even from this trivial example it can be seen that the values are clustered: the values 20..31 are never used in the bit vector.

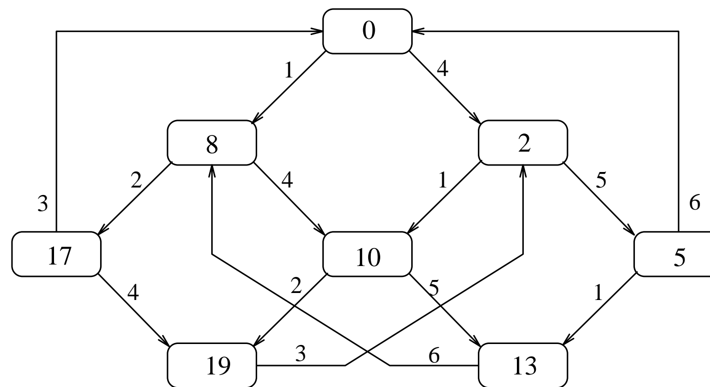


Figure 4: Reachability graph labelled with state values and transition numbers.

Consider model checking the CTL formula $AG(\neg(C_1 \wedge C_2))$. With the enumeration explained above this formula will be invalid if a state exists with either the value 20 or 21. The values 20 and 21 indicate that both processes are in their critical region and the semaphore can either have value 0 (value 20) or 1 (value 21). It can be seen from the graph that this is not the case, thus the formula is satisfied by the model.

2.4 Summary

The model checker described in section 2.3.1 determines whether a given CTL formula is satisfied in the start state of a system and, depending on the result, new states may be generated. Therefore the model checker consists of two parts: an evaluator to compute the truth value of CTL formulae in the current state and a state generator for generating new states by executing transitions. The CTL evaluator guides the state generator to construct the partial reachability graph necessary to determine the truth-value of a given CTL formula.

Since the model checker described here accepts transition systems (a low-level formalism) as input, a higher level validation language is needed to model practical systems. This proved to be the first challenge: to design a higher-level validation language that is natural to use and which can be translated into effective transition systems of equivalent functionality.

Chapter 3

A Validation Language for Reactive Systems

Pnueli defines a reactive system [47] as a system that does not terminate but rather maintains a permanent interaction with its environment. Operating systems, process control systems and communication protocols are examples of reactive systems. The modelling and validation of reactive systems have enjoyed widespread interest [47, 69, 76] but most work, until now, has been done in the area of communication protocols. Operating system kernels, for example, have not enjoyed the same amount of interest. It was therefore decided to concentrate on the validation of a locally developed kernel [36] which supports processes and message passing. This kernel can be seen as a representative example of the kind of reactive system we wish to validate. Model checking was selected as the validation technique to be used.

The input accepted by our model checker is a transition system. However, transition systems are too primitive to model practical systems; a higher-level validation language with the following properties was needed:

- It must be specifically suited for modelling operating system kernels.
  - It should support modular design.
  - Complex data structures, namely *records* and *sequences* must be supported.
  - Efficient control structures must be supported.

  - It must support concurrent processes that can communicate via message passing.

- The language must allow efficient model checking.
  - Structures must have fixed sizes determinable at compile time to allow efficient model checking.
  - It must be possible to translate the language to the transition system format accepted as input by the model checker.

- We wanted the validation language to be adaptable as dictated by either the model checking or kernel modelling experience. This implied designing our own experimental language which was influenced by the successful protocol validation language Promela.

In the next section the experimental validation language *ESML* (Extended State Machine Language) [27], based on the above mentioned design criteria, will be discussed. Examples of models written in ESML are given in section 3.2. In section 3.3 a validation system for reactive systems will be described.

3.1 ESML

Process algebras are widely used as a basis for the design of specification or modelling languages for concurrent systems. Two well-known process algebras are *CCS* [73] and *CSP* [48]. Specification languages based on process algebras include Esterel [5], Lustre [78], LOTOS [56], Argos [69], Statecharts [46] and Promela [51]. CSP has been used as the basis of programming languages such as Occam [55] and Joyce [10]. The design of ESML was inspired by Joyce and Promela. Joyce is a strongly typed programming language for distributed systems based on CSP and Pascal. It provided a design framework for ESML. Promela is a protocol validation language and it showed which language constructs can be implemented efficiently. Promela also served as an example for designing the run-time environment of ESML.

State machines [2] are commonly used to describe concurrent systems [51, 60, 62, 80]. Therefore, extended state machines (ESMs) were chosen as a basic building block to model

reactive systems. An ESM is a state machine enhanced with data variables, assignment instructions, guarded instructions and communication instructions. A model of a reactive system consists of a hierarchy of ESMs. This is similar to the *agents* used in Joyce. An ESM consists of two parts: a declaration part and a body of ESML instructions. The declaration part can contain constant, type and variable definitions as well as other ESMs. It was a design decision to allow only *local* variables: only the ESM in which a given variable is defined can access the variable in its body. ESMs execute concurrently once created and may communicate via channels. Communication is based on the principles of CSP: unbuffered, blocking communication via channels (which are accessed through ports). The design of a reactive system is usually broken down into modules, objects and procedures. In ESML an ESM can simulate each of these constructs. The fundamental concepts of the language will now be discussed.

3.1.1 Type and Variable Definitions

The syntax for constant, type and variable definitions are defined in extended BNF as follows:

```
ConstantDefinitionPart = "CONST" ConstantDefinition { ConstantDefinition }.

TypeDefinitionPart = "TYPE" TypeDefinition { TypeDefinition }.

VariableDefinitionPart = "VAR" VariableDefinition {VariableDefinition }.

ConstantDefinition = ConstantName "=" Constant ";".

Constant = Numeral | "TRUE" | "FALSE" | ConstantName.

TypeDefinition = TypeName "=" NewType ";".

NewType = ListType | RecordType | PortType |
          SubrangeType | EnumType.

SubrangeType = Constant ".." Constant.

EnumType = ConstantName { "," ConstantName }.

VariableDefinition =  VariableName { "," VariableGroup } ":" TypeName ";".
```

It is widely accepted that the use of global variables should be avoided as part of "good programming practice". Therefore, in ESML an experiment was undertaken by excluding

global variables altogether. Thus far we are convinced that the modelling power of ESML was not affected by this decision and the absence of global variables simplified the design of the run-time environment as will be discussed in Chapter 5.

As explained in Chapter 2 the size of the state space is determined by the number of bits in the state vector: if there are n bits in the state vector then there are 2^n potential states in the state space. This implies that the state vector must be kept as small as possible. When a variable is declared it is allocated a number of bits in the state vector as determined by its type. For this reason only the *BOOLEAN* type is predefined. Instead of predefining a number of standard types (say, 16 bits for an integer type), subranges of integers[1] are declared in order to allocate the minimum number of bits. In this respect ESML differs from Promela, where the predefined types *byte* (8 bits), *short* (16 bits) and *int* (32 bits) are used instead of subranges. In the Promela system the global state value is transformed into a 28 bit value by hashing. Therefore, the compaction of the state in ESML is done at compile time while the Promela system does it during run-time.

In ESML the assignment operator "`:=`" can be used to assign new values to variables. The syntax for the assignment instruction is defined by:

```
AssignmentInstruction = VariableAccess ":=" Expression.
```

and the type of `Expression` must be compatible with that of `VariableAccess`.

Constant definitions and enumerated types are also supported. The keywords "`CONST`", "`TYPE`" and "`VAR`" respectively indicate constant, type and variable definitions. The following example illustrates the definitions of a constant, subrange type, enumerated type and variables:

```
CONST max = 10;
TYPE Number = 0..max;
     Colours = red,green,blue;

VAR counter : Number;
    attribute : Colours;
```

[1]In the current implementation of ESML only non-negative integers are allowed and all subranges must start at 0

In the context of these type definitions, the following assignments are valid:

```
counter := counter + 1;
attribute := red;
counter := max;
```

Structured Types

Records and lists are the two structured types supported by ESML. The syntax of structured types in ESML is defined as follows:

```
RecordType = "(" FieldList ")".

FieldList = TupleSection { ";" TupleSection }.

TupleSection = FieldName TupleTail.

TupleTail = "," TupleSection | ":" TypeName.

ListType = "LIST" "[" Constant "]" "OF" TypeName.
```

Lists in ESML are finite sequences numbered from 0 to n where n is the constant specified in the declaration of the list (see grammar above). Lists must be finite since a variable of a list type is stored in the finite state vector. Records are finite by definition. Combinations of these structures such as lists of records, a record of lists, a list of lists etc., are also allowed.

An empty list is indicated by the symbol “`<>`”. A list must be initialised as empty before it can be used. The ESML assignment instruction `list := <>` assigns the empty list to the list variable **list**. The operator “::” is used to concatenate an element to either end of a list. The intrinsic function **HD(list)** returns the first element of **list**. Similarly, **TL(list)** returns the value of **list** without the first element and **LEN(list)** returns the number of elements in **list**.

The following examples illustrate the definition and use of structured types:

```
Definitions:

   TYPE ProcessRecord = (ProcessID, Priority : Number);
        Queue = LIST [10] OF ProcessRecord;

   VAR ProcessQueue : Queue;
       Process : ProcessRecord

Usage:

   ProcessQueue := <>;
   Process.ProcessID := 1; Process.Priority := 0;
   ProcessQueue := ProcessQueue::Process;
   Process := HD(ProcessQueue);
```

Port Types

In the version of CSP introduced in 1978 it was necessary for a process sending a message to name a recipient process explicitly. Silberschatz [81] proposed an alternative to this explicit naming approach: a process should name a *port* through which communication takes place. This idea was adopted in ESML. Two ESMs communicate via a communication channel which they access through port variables in either ESM. The access of channels through the use of ports will be discussed further in section 3.1.4.

Brinch Hansen found that the most common errors in Joyce programs were type errors in communication instructions. He concluded that "any CSP language must include message declarations which permit complete type checking during compilation". For this reason the idea of a *port type*, as defined in Joyce, was adopted. A port type T defines an *alphabet* which is a set $\{s_0(T_0), s_1(T_1), ..., s_n(T_n)\}$ of symbol classes. The values in each symbol class $s_i(T_i)$ are formed by prefixing each value of type T_i with the name s_i. Each value T_i represents a set of messages. The syntax for defining a port type is given below.

```
PortType = "{" Alphabet "}".

Alphabet = SymbolClass { "," SymbolClass }.

SymbolClass = SymbolName [ "(" MessageType ")" ].

MessageType = TypeName.
```

Symbol classes make it possible to group related messages together conveniently. In the example below the symbol classes `value` and `EndStream` indicate that a stream of messages

may contain values of type `Number` or a special termination message. `EndStream` is an example of a symbol class with no associated message type and is called a *signal*.

```
TYPE Number = 0..10;
     Stream = {value(Number), EndStream};

VAR channel : Stream;
```

Promela supports synchronous and asynchronous communication. The principles of CSP were used as a guideline for the design of ESML and thus communication is simpler: only unbuffered synchronous message passing is supported. In CSP a channel may only be defined between two processes. In ESML this restriction was removed and a channel for message passing can be defined between two or more ESMs. If more than two ESMs are ready to communicate on the same channel, it may be possible to match them in several different ways. However, the actual synchronisation is binary: only one sender and one receiver ESM can communicate at a time. Thus multi-way synchronisation [6] as in LOTOS is not supported. Multi-way synchronisation means that more than one ESM can receive the same message from a specific sender ESM.

The sending and receiving of messages between two ESMs are accomplished by using two simple communication instructions. A message m of type T which belongs to symbol class S of alphabet A is sent via a port p by the instruction `p!S(m)`. Similarly a message m which belongs to symbol class S is received via a port p and assigned to a variable x of type T by writing `p?S(x)`. When a message is sent, the sending ESM waits until a matching receive instruction is executed by another ESM. Similarly an ESM wanting to receive a message waits until a message of the specified type is sent. The syntax for communication instructions is:

```
CommunicationInstruction = VariableAccess "!" SymbolName [ "(" Expression ")" ] |
                           VariableAccess "?" SymbolName [ "(" VariableAccess ")" ].
```

Type Equivalence

In ESML type checking is based on *name equivalence*. Types `int1 = 0..10` and `int2 = 0..10` are thus not type compatible. Although it is possible, the intention is usually not

to define several different subranges in the same model. A single subrange of integers that is suitable for all numbers in a model is usually convenient.

3.1.2 Expressions

Arithmetic expressions based on the four basic operators, *addition* ("`+`"), *subtraction* ("`-`"), *multiplication* ("`*`") and *integer division* ("`DIV`") are supported. Boolean expressions with *and* ("$\wedge$"), *or* ("$\vee$"), *not* ("`~`") and the usual relational operators ("`=`", "`#`", "`>`", "`<`", "`<=`", "`>=`") are also supported. After evaluating an expression the result can be assigned to a type compatible variable. Range checks are always done before assignments.

3.1.3 Control Structures

Dijkstra guarded commands [29] provide the control structure for ESML. A guarded command has the form $Guard \rightarrow InstructionSeq$. A guard is a boolean expression. If the guard evaluates to true the instruction sequence is executed. If more than one guard is true at the same time a nondeterministic choice must be made between the true guards. The syntax of the nondeterministic alternative (IF) and repetitive (DO) commands is defined by:

```
IfConstruct = "IF" GuardedCommandList "END".

DoConstruct = "DO" GuardedCommandList "END".

GuardedCommandList = GuardedCommand { "[]" GuardedCommand }.

GuardedCommand = Expression "->" InstructionSeq.

InstructionSeq = Instruction { ";" Instruction }.

Instruction = AssignmentInstruction | IfConstruct |
              DoConstruct | PollConstruct   |
              "SKIP" | ESMInstruction |
              CommunicationInstruction.
```

The following examples illustrate the syntax of the IF and DO constructs. Note that in the example of the IF construct both the guarded commands can be executed when $x = 10$ and therefore a nondeterministic choice must be made.

```
IF x >= 10 -> z := x
[] x <= 10 -> z := x+10
END;

DO c > 0 -> c := c - 1
[] c = 0 -> c := 1000
END;
```

The meaning of IF and DO structures can be described as follows:

IF: When a guard is true and its sequence of instructions has been executed the IF terminates. When all the guards are false the IF terminates with an error.

DO: When a guard is true and its sequence of instructions has been executed the list of guards are evaluated again. Termination occurs when all the guards are false.

The use of communication instructions in the guard has been studied extensively in the literature [3, 30, 54, 86]. In CSP when all the processes named by input guards terminate a repetitive command also terminates. However, Hoare later declared this convention to be "complicated to define and implement" [48, 83, 86].

Promela allows input and output instructions in the guards. Promela avoids the problem of termination of guarded commands by providing mechanisms ("break" and "goto") to terminate IF and DO constructs. Unlike in Dijkstra guarded commands the IF and DO in Promela are repeated when all their guards are false. In Promela a DO can only be terminated by one of the termination instructions ("break" or "goto"). An IF terminates when one of its guarded commands is executed. In the Promela code

```
    do
    :: in?data -> out!data
    :: in?error -> goto err
    :: in?stop -> break
    od
err:...
```

when an *error* or a *stop* message is received on the *in* channel the DO is terminated by a *goto* or a *break* instruction respectively. In both these cases the execution continues with the first instruction after the DO. In the general case the *goto* can allow execution to

continue at any instruction and the *break* will always continue with the first instruction after the DO. Therefore, the termination semantics of the IF and DO in Promela differs from the semantics of these constructs in Dijkstra guarded commands: if all the guards of an alternative command are false then it is an error and for a repetitive command the command terminates. In ESML the semantics of Dijkstra guarded commands is closely followed. For this reason we adopt the Brinch Hansen approach (as in Joyce) for implementing input and output instructions in guards: when communication guards are needed they must be used within a new nondeterministic construct called a *POLL*. No input or output guards are allowed in an IF or DO. A POLL is defined as follows:

```
PollConstruct = "POLL" GuardedPollList "END".

GuardedPollList = GuardedPollCommand { "[]" GuardedPollCommand }.

GuardedPollCommand = PollGuard "->" InstructionSeq.

PollGuard = CommunicationInstruction [ "/\" Expression ].
```

The guard of a POLL consists of the conjunction of two parts namely a communication instruction and an optional boolean expression. The communication will only be executed if the boolean expression is true (if not omitted). The boolean expression may contain a variable just assigned by an accompanying input instruction (see the next example below: `x` will only be received if `x > 0`). The meaning of a POLL construct is:

- Only guards consisting of communication instructions and an optional expressions are allowed. When a communication instruction is matched and the optional boolean expression is satisfied, the associated instruction sequence is executed and the POLL terminates. The guards are continually evaluated until one is true.

The example below illustrates the use of the POLL construct. The POLL consists of two guards: the first is an input instruction for the symbol class `value` on channel `in` if the receiving variable `x` is greater than zero and the second guard is an output instruction sending a message `0` of symbol class `value` on channel `out`.

```
POLL in?value(x) /\ x > 0 -> x := x*x;
                             out!value(x + y)
[]   out!value(0) -> SKIP
END;
```

With the addition of the POLL construct, instead of termination instructions (like in Promela), the termination conditions for the DO and IF becomes straightforward. Let $g_1, g_2 \ldots, g_n$ be the n guards of either a DO, IF or POLL. The termination condition of a DO is then:

$$(\neg g_1) \wedge (\neg g_2) \wedge \ldots \wedge (\neg g_n)$$

The termination conditions for an IF and POLL are the same, namely

$$g_1 \vee g_2 \vee \ldots \vee g_n$$

3.1.4 ESMs : Definition and Activation

An ESML model consists of a hierarchy of nested ESMs. The outermost ESM is activated first. New ESMs that execute concurrently with their creators are activated by executing ESM activation instructions. Recursive activation is also supported: an ESM can activate another instance of itself. An ESM terminates only when all ESMs activated by it have terminated. The syntax of an ESM activation instruction is:

```
ESMInstruction = ESMName [ "(" ActualParameterList ")" ].
```

The syntax for the definition of an ESM is defined by the following productions.

```
ESM = "ESM" ESMName Block ESMName

Block = [ "(" FormalParameterList ")" ] ";" ESMBody.

FormalParameterList = ParameterDefinition { ";" ParameterDefinition }.

ParameterDefinition = "IN" VariableGroup |
                      "OUT" VariableGroup |
                       VariableGroup.

ESMBody = [ ConstantDefinitionPart ] [ TypeDefinitionPart ]
          [ VariableDefinitionPart ] { ESM ";" }
          "BEGIN" InstructionSeq "END".
```

ESMs may have two kinds of parameters: value parameters and port parameters. An ESM activation instruction must provide an actual parameter (of matching type) for every formal parameter defined by the ESM definition.

For value parameters the value of the corresponding actual parameter is assigned to each formal parameter. Value parameters may not be used on the left hand side of assignments.

Whenever a variable of a port type is declared it constitutes a channel creation and is called a channel variable. Passing the channel variable on to other ESMs as an actual parameter allow these ESMs to communicate on the channel via the local port parameter. The ESM that declared the channel variable can also communicate via the channel. Using a port parameter to *receive* a message when it was intended for *sending* messages or vice versa was found to be a common error. For this reason port parameters are marked with one of the keywords “IN” or “OUT” to indicate the direction of the message transfer.

```
ESM Init;

TYPE int = 0..5;
     Alphabet = {Message, value(int)};


VAR channel : Alphabet;

  ESM A(OUT Chan : Alphabet;x : int);
  BEGIN
    Chan!Message;
    Chan!value(x)
  END A;

  ESM B(IN Chan : Alphabet);
  VAR x : int;
  BEGIN
    Chan?Message;
    Chan?value(x)
  END B;

BEGIN
  A(channel,2);B(channel)
END Init;
```

In the example above `channel` (in `ESM Init`) is a channel variable. By passing `channel` to `ESM A` and `ESM B` as actual parameters a communication channel is created between `ESM A` and `ESM B`. `Chan` (in `ESM A`) is a port parameter indicating that messages may only be sent via the channel and `Chan` (in `ESM B`) is a port parameter indicating only incoming messages can be received on the channel.

3.1.5 Scope Rules

The scope of a variable α in an ESM A extends from directly after its declaration to the end of A. However, α is unknown in any ESM contained in A because global variables are not allowed. The scope rules for constants and type definitions are different. Constants and types are known from directly after their declaration to the end of the ESM containing the declaration unless they are redefined by some nested ESM. To allow recursion an ESM name is known from directly after its declaration until the end of the ESM containing the definition. The ESM declaration includes the possible parameters associated with it. No forward references are allowed and therefore mutually recursive activation is excluded. Recursion is of limited value in ESML and models relying on mutual recursion were considered too complex to consider. However, this restriction may perhaps be removed in future.

3.1.6 Requirement Specifications

The requirement specification of an ESML model is given by a CTL formula. This formula must hold in the model of the system described by the ESML code. The complete validation model consists of an ESM (which may contain nested ESMs) followed by a requirement specification that must hold in the ESM. A model checker is used to determine whether a CTL formula holds in a model.

A validation model can now be defined:

```
ValidationModel = ESM ";" Requirement.

Requirement = "ASSERT" CTLFormula.
```

A CTL formula contains propositions in the form of boolean expressions that can either be true or false in the ESML model. These expressions can contain any variable in the ESML model except channel variables (accessed via ports). Channel variables are excluded because communication is unbuffered and synchronous and therefore channel variables can never store any messages. No proposition can thus be defined on channel variables. A proposition may also contain the *HD*, *TL* and *LEN* operators for lists. The syntax for CTL formulae is defined by the following grammar.

```
CTLFormula = SubFormula |
             SubFormula "/\" CTLFormula |
             SubFormula "\/" CTLFormula |
             SubFormula "=>" CTLFormula.

SubFormula = "AG" CTLFormula | "EG" CTLFormula |
             "AF" CTLFormula | "EF" CTLFormula |
             "AX" CTLFormula | "EX" CTLFormula |
             "A" "(" CTLFormula "U" CTLFormula ")" |
             "E" "(" CTLFormula "U" CTLFormula ")" |
             "~" CTLExpression | CTLExpression.

CTLExpression = "(" CTLFormula ")" | Proposition.

Proposition = CTLVarAccesss RelationalOperator RightSide |
              CTLVarList RelationalOperator RightSide |
              "TRUE" | "FALSE".

RelationalOperator = "<" | "<=" | "=" | "#" | ">=" | ">".

CTLVarAccess = ESMName "." Tail.
Tail = CTLVarAccess | VariableAccess.
RightSide  = Numeral | CTLVarAccess | CTLVarList.
CTLVarList = "HD" "(" CTLVarAccess ")" |
             "TL" "(" CTLVarAccess ")" |
             "LEN" "(" CTLVarAccess ")".
```

A variable contained in a proposition must be uniquely named in the formula. If a variable x is declared within an ESM B which is nested within ESM A, the variable must be named "$A.B.x$". This ensures uniqueness of each variable in the formula. Examples of requirement specifications will be given in section 3.2.

3.2 ESML Examples

The model given in Figure 5 illustrates a recursive implementation of a factorial algorithm. The `ESM Fact` models a function that recursively determines the factorial of the cardinal value `n` with the result of each level being returned via the `child` channel to the previous level. The CTL formula checks that the computation of the factorial of 6 was correct, by stating that the result will always be 720. During execution `ESM Factorial` is initially executed and activates `ESM Fact` with the value parameter 6 and the `child` channel. `ESM Factorial` then blocks while waiting for a message on the `child` channel which it assigns to its `result` variable on reception. `ESM Fact` recursively activates itself with the value of its value parameter (`n`) decremented by one and its local `child` channel. When the value

of `n` becomes 1 there are six activations of `Fact` and one activation of `Factorial` executing concurrently. Each `Fact` ESM then accepts the result of the previous level via the `child` channel and outputs its result via the channel accessed by the port parameter `p`.

```
ESM Factorial;
TYPE Int = 0..1000;
     ResultChan = {Res(Int)};

VAR  result : Int;
     child : ResultChan;

ESM Fact(n : Int;OUT p : ResultChan);
VAR result : Int;
    child : ResultChan;
BEGIN
  IF n <= 1 -> p!Res(1)
  [] n >= 2 -> Fact(n-1,child);
               child?Res(result);
               p!Res(n*result)
  END
END Fact;

BEGIN
  Fact(6,child);
  child?Res(result)
END Factorial;

ASSERT
  AF(Factorial.result = 720)
```

Figure 5: An ESML model of a recursive factorial algorithm

The ESML model in Figure 6 represents a version of the *Alternating bit* protocol [20, 51]. The data messages `dm0` (`dm1`) are sent by `ESM Sender` and on reception the receiver `ESM Receiver` acknowledges reception of `dm0` (`dm1`) by sending messages `am0` (`am1`). A transmission error is simulated by sending an error message `err`. Whenever a message is to be sent a nondeterministic choice is made between sending the correct message or the error message. The sender is in state `Gendm0` (`Gendm1`) when it has just generated the message `dm0` (`dm1`) and is about to send it to the receiver. When the `dm0` (`dm1`) message is received the receiver enters the state `Acptdm0` (`Acptdm1`), which indicates that the message was successfully accepted. The CTL specification expresses the following liveness property (section 2.1.2): whenever the message `dm0` is sent, it will eventually be received.

Comments are delimited by "`(*`" and "`*)`".

```
ESM Atb;
TYPE Send = {dm0,dm1,err};  Recv = {am0,am1,err};
     Loc = Gendm0,Gendm1,Acptdm0,Acptdm1;
VAR Snd : Send; Rcv : Recv;

  ESM Sender(IN Rcv : Recv; OUT Snd : Send);
  VAR more : BOOLEAN; state : Loc;
    BEGIN
    DO TRUE ->
      state := Gendm0;   (* Generate dm0 *)
      POLL Snd!dm0 -> SKIP [] Snd!err -> SKIP END;
      more := TRUE;
      DO more ->
        POLL Rcv?am0 -> more := FALSE [] Rcv?am1 -> Snd!dm0 [] Rcv?err -> Snd!dm0 END
      END;
      state := Gendm1;   (* Generate dm1 *)
      POLL Snd!dm1 -> SKIP [] Snd!err -> SKIP END;
      more := TRUE;
      DO more ->
        POLL Rcv?am1 -> more := FALSE [] Rcv?am0 -> Snd!dm1 [] Rcv?err -> Snd!dm1 END
      END
    END
  END Sender;

  ESM Receiver(IN Snd : Send; OUT Rcv : Recv);
  VAR more : BOOLEAN; state : Loc;
  BEGIN
    DO TRUE ->
      more := TRUE;
      DO more ->
        POLL Snd?dm0 -> more := FALSE [] Snd?dm1 -> Rcv!am1 [] Snd?err -> Rcv!am1 END
      END;
      state := Acptdm0;  (* Accept dm0 *)
      POLL Rcv!am0 -> SKIP [] Rcv!err -> SKIP END;
      more := TRUE;
      DO more ->
        POLL Snd?dm1 -> more := FALSE [] Snd?dm0 -> Rcv!am0 [] Snd?err -> Rcv!am0 END
      END;
      state := Acptdm1;   (* Accept dm1 *)
      POLL Rcv!am1 -> SKIP [] Rcv!err -> SKIP END
    END
  END Receiver;

BEGIN
  Sender(Rcv,Snd); Receiver(Snd,Rcv)
END Atb;

ASSERT AG((Atb.Sender.state = Atb.Gendm0) => AF(Atb.Receiver.state = Atb.Acptdm0))
```

Figure 6: An ESML model of the alternating bit protocol

3.3 A Validation System

The increasing complexity of software systems has prompted the use of formal methods in the development of such systems. Formal methods not only give software development a scientific foundation, but also helps with the elimination of potential errors. However, the acceptance of formal methods is largely dependent on the availability of powerful tools. An integrated tool environment that assists the practitioner in creating, manipulating, and understanding a formal specification is of critical importance.

Integrated tool environments already exist and examples of such systems that support model checking as validation tool include (the validation language accepted by each system appear in parentheses): LOEWE (LOTOS) [61], XESAR (ESTELLE/R) [44], The Concurrency Workbench (CCS) [24], SPIN (Promela) [51], SMV System (SMV) [71] and ARGONAUTE (ARGOS) [69]. All these systems show that the control structure of models can be easily model checked. However, the data part of models is more difficult to model check and is at the moment a subject that receives much attention. ESML models can have complex data structures (records and lists) and in Chapters 5 and 6 it will be shown how these models can be model checked by using compaction techniques on the data structures.

Figure 7 shows the structure of the completed validation system as currently planned. It is meant to support the design and validation of reactive systems. ESML is used as validation language and model checking as a validation tool.

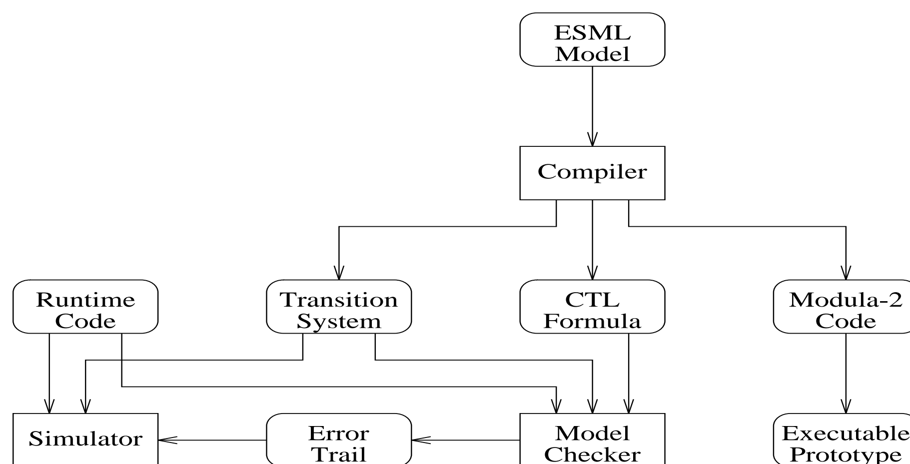


Figure 7: The proposed validation system

The completed system will consist of an ESML compiler, simulator and model checker. It will also support the direct translation of ESML to Modula-2 code. The Modula-2 code can be compiled and executed to serve as a prototype of the intended system.

For model checking the compiler generates an internal representation of the given CTL formula together with a transition system generated from the ESML model. The transition system and the CTL formulae are both represented in Modula-2 code which is compiled and linked together with the model checker code. The transitions are executed in a specific run-time environment during the model checking procedure. Whenever the model checker invalidates a CTL formula in a certain state it produces an error trail to indicate how this erroneous state can be reached.

The simulator will allow the user to execute an ESML model. The user can observe the contents of variables and set breakpoints in the ESML code during the simulation. The same transition system (represented in Modula-2) used by the model checker is used as input to the simulator. The transition system is compiled and linked together with the simulator code to form an executable simulator for the given ESML model. The simulator can also take the error trail produced by the model checker as input. This allows the user to observe the execution of the ESML model for reaching the erroneous state.

The difference between the execution of the simulator and the model checker is that the simulation is guided by the user and model checking executes a system according to a CTL formula.

This validation system can thus be used for the entire design cycle of a reactive system:

- A model which captures the behaviour of a reactive system can be expressed in ESML.
- The behaviour of the system can be analysed by using the simulator.
- When the user is satisfied with the structure and behaviour of the reactive system, correctness requirements of the system can be expressed as CTL formulae. The system can then be model checked against its requirements specifications. Analysing possible error trails with the simulator will result in further improvements to the reactive system being made. If no more errors are reported by the model checker the

model satisfies its requirements.

- The ESML model of the reactive system can now be translated into equivalent Modula-2 code. This code can be compiled to serve as a prototype of the intended system.

The model checker has been completed and is described in Chapter 2 and [26]. The simulator is still being developed but a simple facility to analyse error trails has been implemented. The facility for direct translation of ESML code to Modula-2 is still in the design phase. A graphical user interface for manipulating the different components of the validation system is also being designed. The implementation of the ESML compiler is described in this thesis. More specifically the rest of the thesis is devoted to the following aspects of the validation system:

- The ESML compiler generating a transition system from an ESML model.
- The run-time environment of the transitions during the model checking of ESML models.
- Optimisations to the run-time environment and transition system to ensure more efficient model checking.

In the next chapter the format of the transitions and internal representation of CTL formulae produced by the compiler will be described.

Chapter 4

The Transition System

In the previous chapter the ESML validation language was introduced. This language is used to define models of systems which can be validated by the model checker described in Chapter 2. ESMs define a modular structure for an ESML model with each ESM exhibiting a specific flow of control throughout its existence. Although the transition system described in section 2.2 could provide a basis for executing ESML models, control information is exploited to create a more efficient transition system.

The transition system defined by Holzmann to represent Promela validation models [51] could be adapted to the requirement of ESML. The format of the transition system is described here, while a run-time environment for executing transitions is the topic of Chapter 5.

The transition system and CTL formula for each specific model are represented by a Modula-2 module. This module is generated by the ESML compiler discussed in Chapter 6. The module is compiled and linked to the model checker code to produce an executable program. When executed, this program checks whether the CTL formula holds in the reachability graph generated from the transition system. As an example the Modula-2 code representing a specific ESML model and CTL formula is shown in section 4.3.

4.1 The Transition Format

A transition is generated for every assignment, ESM activation, SKIP, input and output instruction in ESML. The guards of an ESML control structure are considered to be executable because guards containing input instructions can change the state of the system being modelled. Therefore a transition is generated for each guard in a control structure. All transitions forming part of each ESM are grouped together and are uniquely identified by numbering them relative to ESMs.

Transitions are executed in the run-time environment described in the next chapter. Every transition has the format: "**IF** guard **THEN** action **END**". A transition is *executable* if and only if its guard evaluates to true in the current state. Otherwise it is said to *fail*. As explained in Chapter 2 the system state is represented by a vector of bits called the *state vector*. Part of the state vector is allocated to each ESM in a model to store the values of its variables. In addition to the data variables of an ESM it also has a location variable indicating the current point of execution in the ESM. In the action part of a transition the location variable is updated to indicate which transition must be executed next. For example the transition code generated for the assignment `x:=10` is:

IF TRUE **THEN**
  SetVar(state,start_x,bitlength_x,10);
  SetVar(state,start_{locvar},$\text{bitlength}_{locvar}$,1)
END

The first SetVar[1] assigns the value 10 to the part of the state vector allocated to variable x. The second SetVar assigns the value 1 to the location variable of the current ESM, to indicate the next transition to be executed.

Because an ESM and a *process* in Promela are similar in structure the transition system used to represent Promela models in the SPIN validation tool was used as a basis for the transition system described here. The transition system used to represent Promela models is in the form of a transition table (matrix). It was decided to implement a similar transition table to represent ESML models, each entry in the table being referenced by an

[1]The SetVar procedure is discussed in the next chapter.

ESM and transition number. Each transition entry has the following fields:

Next Transition: This field indicates the next transition to be executed.

Next Guard: If a transition represents an ESML guard, this field indicates which transition to execute when the current transition fails.

Code: A pointer to the code associated with this transition. In the present context ESML code fragments will be used in this field to represent the Modula-2 code. In section 4.3 it will be shown that this field is an index into a **CASE** structure where the code is stored.

In the remainder of this section the *format* of transition entries will be described.

4.1.1 Fundamental Instructions

Consider the following sequence of ESML instructions:

```
ESM Test(IN in : alpha1; OUT out : alpha2);
VAR x,y : int;
BEGIN
  Example(x,y);
  x := 5;
  SKIP;
  out!value(5);
  in?value(x)
END Test;
```

The following transition entries will be generated for this code fragment (assuming the ESM number of `ESM Test` is 1):

ESM Number	Number	Next Transition	Code	Next Guard
1	0	1	$Example(x, y)$	...
1	1	2	$x := 5$	...
1	2	3	$SKIP$	...
1	3	4	$out!value(5)$	...
1	4	5	$in?value(x)$	...
1	5	...	$Termination$	...

Note that the **Next Guard** field is empty for all fundamental instructions. The last transition of each ESM is a *termination* transition. Being the last transition, its **Next Transition** field is empty.

4.1.2 Guarded Commands

The only difference between transition entries generated for the guards of a control structure and the fundamental instructions is that the **Next Guard** field is used to indicate the transition for the following guard in the control structure. The transition indicated by this field will be executed if the current guard fails. The **Next Transition** field indicates the transition entry generated to represent the instruction sequence associated with each guard in a control construct.

For run-time checking of the termination conditions of control structures a *control* transition entry is generated after the transition entry for the last guard in the structure. The **Next Guard** field of the entry for the control transition indicates the kind of control structure being executed. The termination conditions for a DO, IF or POLL are controlled by the state generator. The code for the control transition of a DO, IF and POLL will respectively perform the following actions:

DO — Terminate the DO and execute the first transition after the DO, since all the guards were false.

IF — Terminate the IF and generate a run-time error, since at least one of the guards of an IF must be true.

POLL — The current ESM cannot execute unless a communication can be executed to match one of the guards in the POLL therefore another ESM must be scheduled for execution.

It will now be explained how the combination of the **Next Transition** and **Next Guard** fields in the transition entry capture the control flow of the DO, IF and POLL constructs.

DO

The following is an example of a DO and the corresponding transition entries generated for it: ("DO" in the **Next Guard** field of control transition entry 5 indicates that this is a DO construct and the ESM Number field is not included because all the transitions belong to the same ESM):

```
x := 10;
DO x > 0 -> x := x - 1
[] x = 0 -> x := 10
END;
y := 5;
...
```

Number	Next Transition	Code	Next Guard
0	1	$x := 10$	...
1	2	$x > 0$	3
2	1	$x := x - 1$	...
3	4	$x = 0$	5
4	1	$x := 10$	...
5	6	...	DO
6	7	$y := 5$	...

A DO construct terminates when all guards are false. In the transition system this is accomplished by setting the **Next Transition** field of the transition entry corresponding to the last instruction in each associated instruction sequence to the first guard transition of the DO. In the example above this is illustrated by the **Next Transition** value 1 in transition entries 2 and 4. Similarly the **Next Transition** field of the control transition (entry 5 in the above example) indicates the first transition after the DO construct (transition entry 6) to be executed.

IF

The following is an example of an IF and its corresponding transition entries:

```
IF x > 0 -> x := x - 1;
            y := x DIV 2
[] x = 0 -> x := 10
END;
y := x;
...
```

Number	Next Transition	Code	Next Guard
0	1	$x > 0$	3
1	2	$x := x - 1$	...
2	6	$y := x\ DIV\ 2$	...
3	4	$x = 0$	5
4	6	$x := 10$	...
5	...	...	IF
6	7	$y := x$	...

An IF terminates if one of its guards is true and the corresponding instruction sequence has been executed. This is accomplished by assigning the **Next Transition** field of the entry for the last transition in an instruction sequence the number of the first transition after the IF. In the example above this illustrated where both transition entries 2 and 4 indicate transition 6 in their **Next Transition** field. If all the guards are false, an IF terminates with a run-time error. Hence no transition will be executed after the control transition of an IF (entry 5 in the above example) and the **Next Transition** field of the control transition entry is empty.

POLL

The following is an example of a POLL and its corresponding transition entries:

```
POLL in?value(x)  -> x := x - 1;
                     y := x DIV 2
[]   out!value(y) -> x := 10
END;
y := x;
...
```

Number	Next Transition	Code	Next Guard
0	1	$in?value(x)$	3
1	2	$x := x - 1$	...
2	6	$y := x\ DIV\ 2$	...
3	4	$out!value(y)$	5
4	6	$x := 10$	...
5	0	...	POLL
6	7	$y := x$	...

As explained in section 3.1.3 a POLL will continually evaluate its guards until one is satisfied and the associated instruction sequence has been executed. To accomplish this, the **Next Transition** field of the entry for the transition corresponding to the last instruction in each associated instruction sequence is set to the first transition after the POLL. In the above example this is represented by the value 6 in the **Next Transition** field of transition entries 2 and 4. Repeated evaluation of the guards is accomplished by setting the **Next Transition** field of the control transition entry to point to the first guard transition of the POLL. In the example this is indicated by the value 0 in the **Next Transition** field of the control transition entry 5.

4.2 CTL formulae

The CTL formula in an ESML model (section 3.1.6) is translated to an internal format for use by the model checker. Trees are used to represent CTL formulae, propositions forming the leaf nodes. These propositions are translated to equivalent code to be executed during model checking. When the code is executed the truth value of the proposition is returned, therefore the format of the code for a proposition (Prop) is: "**RETURN** Prop". For each proposition in the leaf nodes of the tree a code pointer is used to locate the code generated for the proposition. The following is a CTL formula with its corresponding tree stored in a table format. The values "p" and "q" in entries 2 and 4 of the table represent pointers to the Modula-2 code used to evaluate the corresponding propositions.

```
AG(p => AF(q))
```

Number	Modality	Left	Right
0	AG	1	...
1	=>	2	3
2	p	...	...
3	AF	4	...
4	q	...	...

4.3 The Modula-2 Transition System

The transition system can be implemented as a table of transitions with the transition code of each entry being executed by an interpreter. In general interpreted code executes slower than native code. Therefore it was decided to generate Modula-2 code for constructing the transition table as well as the code to be executed for each entry. This Modula-2 code is compiled and linked to the model checker to form an executable model checker. In the SPIN system [51], Holzmann use a similar technique by generating C code representing a Promela model. Generating the Modula-2 code is done in two phases:

- Firstly, code is generated to construct the transition entries with an index (code pointer) in each entry indicating the transition code to execute for the transition.
- Secondly, the transition code is generated and inserted into a **CASE** instruction with the index in a transition entry indicating a label in the CASE.

The transition table is generated by a set of calls to the *MakeTrans* procedure which generates one entry for each transition in the system.

$$MakeTrans(Esm, Trans, NextTrans, Code, NextGuard)$$

During the initialisation phase of model checking each of these calls to the *MakeTrans* procedure are executed to generate the entries in the transition table.

To preserve modularity every ESM is represented by a procedure which contains a CASE with only the transition code for that ESM (see the example below). This procedure is executed whenever a transition in the ESM is to be executed. If the guard of the transition

is true the procedure returns the new state generated by executing the transition's action. The procedure is passed the current state and the index into the CASE statement (code pointer in the transition entry) as parameters.

As an example consider the following ESML code with its corresponding transitions and code in Modula-2.

```
ESM Example;
TYPE int = 0..10;
VAR x : int;
BEGIN
  x := 10;
  DO x > 0 -> x := x - 1
  [] x = 0 -> x := 10
  END
END;
```

The following calls to *MakeTrans* initialise the transition system to the format in the table below.

```
  MakeTrans(0,0,1,1,empty);
  MakeTrans(0,1,2,2,3);
  MakeTrans(0,2,1,3,empty);
  MakeTrans(0,3,4,4,5);
  MakeTrans(0,4,1,5,empty);
  MakeTrans(0,5,6,empty,DO);
  MakeTrans(0,6,empty,6,empty);
```

ESM	Number	NextTrans	Code	NextGuard
0	0	1	1	...
0	1	2	2	3
0	2	1	3	...
0	3	4	4	5
0	4	1	5	...
0	5	6	...	DO
0	6	...	6	...

The transition code for the transitions in the table above is given in the procedure `ESMExample`.

```
PROCEDURE ESMExample(VAR state : StateVector;
                         Index : CARDINAL);
BEGIN
   CASE Index OF
   1: IF TRUE THEN
        SetVar(state,start_x,bitlength_x,10); (* change value of x *)
        SetVar(state,start_locvar,bitlength_locvar,1) (* update location variable *)
      END
  |2: IF (GetValue(state,start_x,bitlength_x) > 0) THEN (* retrieve and test value of x *)
        SetVar(state,start_locvar,bitlength_locvar,2)
      END
  |3: IF TRUE THEN
        expr := GetValue(state,start_x,bitlength_x) - 1;
        SetVar(state,start_x,bitlength_x,expr);
        SetVar(state,start_locvar,bitlength_locvar,1)
      END
  |4: IF (GetValue(state,start_x,bitlength_x) = 0) THEN
        SetVar(state,start_locvar,bitlength_locvar,4)
      END
  |5: IF TRUE THEN
        SetVar(state,start_x,bitlength_x,10);
        SetVar(state,start_locvar,bitlength_locvar,1)
      END
  |6: (* Remove ESM Example from the list of active ESMs *)
   END
END ESMExample;
```

The procedure `SetVar(state,start_x,bitlength_x,expr)` changes the value of variable `x` in the state vector to the value of `expr`. `GetValue(state,start_x,bitlength_x)` retrieves the value of variable `x` from the state vector.

The tree format for the CTL formula is generated in a similar fashion. The procedure `MakeCTLTree(Operator,left,right)` takes as input a number corresponding to a CTL operator or a proposition as well as the left and right subtree number. If the `Operator` number indicates a proposition the number is used to index the transition for that proposition in a CASE structure which contains all the propositions in the CTL formula.

The following calls to the procedure `MakeCTLTree` will generate the CTL tree (in table format) for the formula $AG((A.x = 1) => AF(A.x > A.y + 2))$:

```
MakeCTLTree(AG,1,empty);
MakeCTLTree(Imply,2,3);
MakeCTLTree(1,empty,empty);
MakeCTLTree(AF,4,empty);
MakeCTLTree(2,empty,empty);
```

Number	Modality	Left	Right
0	AG	1	...
1	=>	2	3
2	1	...	...
3	AF	4	...
4	2	...	...

The following transition code is generated in a CASE structure for the propositions $A.x = 1$ and $A.x > A.y + 2$. The CASE is contained in the `EvaluateProposition` procedure.

```
PROCEDURE EvaluateProposition(state : StateVector;
                              Index : CARDINAL) : BOOLEAN
VAR start : CARDINAL;
BEGIN
   CASE Index OF
   1: RETURN GetValue(state,start_x,bitlength_x) = 1
  |2: RETURN GetValue(state,start_x,bitlength_x) > (GetValue(state,start_y,bitlength_y)+2)
   END
END EvaluateProposition;
```

In summary, the module representing the transition system and CTL formula contains four procedures: `ESMExample`, `EvaluateProposition`, `InitTransitionSystem` and `InitCTLformula`. During initialisation the model checker executes the procedures `InitTransitionSystem` and `InitCTLformula` which then respectively execute a number of calls to `MakeTrans` to build the transition table and `MakeCTLTree` to build the CTL tree.

On closer inspection it was discovered that the transition table is redundant because all the control flow information can be updated when the transition code is executed. For instance the location variable of an ESM, after executing a transition, has the same value as the **Next Transition** field for the transition entry. The storage reduction obtained by this optimisation will not significantly reduce memory requirements for model checking because the storage requirement for the transition table is small compared to that for the bit vector.

The ESML code fragments used in this example were chosen not to include ESM activation, input, output or instructions manipulating structured data. The reason for this is that these (more complex) instructions are executed within a specific execution environment. This run-time environment is the topic of the next chapter.

Chapter 5

A Run-Time Environment for ESML

Due to the complexity of high-level validation languages, models written in such languages cannot be validated directly. An intermediate representation, such as a transition system, is required that can be efficiently interfaced with a model checker. Therefore, a compiler is required to generate a transition system from a high-level model. In our case the source language is ESML and the target language the transition system described in Chapter 4. During model checking the state generator executes transitions to generate the partial reachability graph necessary to determine the truth value of a given CTL formula. Therefore, in addition to the compiler an efficient run-time environment is required for executing transitions.

This chapter will focus on the run-time environment which consists of two parts:

- Storage organisation during the execution of transitions.
- State generation and CTL formula evaluation during model checking.

Storage organisation (section 5.1) is centered around operations performed on the data structures in the run-time environment. In section 5.2 state generation during model checking is described. Evaluating CTL formulae is discussed in section 5.3. A compiler which translates ESML models to functionally equivalent transition systems will be described in Chapter 6.

The structure of the model checker and transition system as well as the key data structures

are shown in Figure 8.

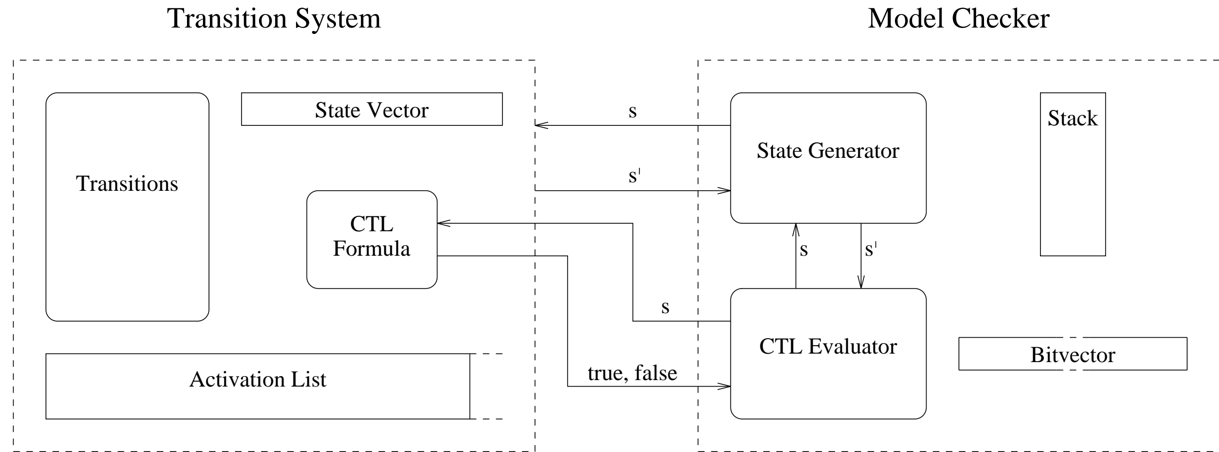


Figure 8: The structure of the model checker and transition system

The model checker consists of two parts: the state generator and the CTL evaluator. The CTL evaluator evaluates a given CTL formula in the current state s and, depending on the result, prompts the state generator to generate a successor state s' of s. The state generator executes an enabled transition in the current state s to generate a new state s'. The ESML compiler generates a transition system from an ESML model. Internally the transition system consists of transitions and a CTL formula.

The key data structures in the run-time environment are the *state vector* and the *run-time activation list*. The current values of the variables of each ESM are stored in the state vector. The system state is determined by the values stored in the state vector. To minimise memory requirements, the minimum number of bits is allocated for each variable in the state vector. When an ESM is activated an activation record is generated to store run-time information about the ESM. Each activation record is linked to the back of the activation list.

As explained in Chapter 2, the key data structures in the model checker are the *bit vector* and the *stack*. The bit vector is used to indicate the unique states generated during the model checking procedure. The current execution path of states is stored in the stack and is used to resolve nondeterminism, as will be explained in section 5.2.2. The stack is also used to detect cycles in the current execution path which may lead to *unfair* execution. Fairness will however not be discussed here; the reader is referred to [26] for a detailed

description of the fairness issues that arise during model checking.

5.1 Storage Organisation

The run-time environment supports synchronous communication between ESMs, activation and termination of ESMs and manipulation of data variables.

Communication can be implemented by respectively appending messages to and removing them from a *channel queue* allocated to each channel. During activation and termination of ESMs, activation records are respectively added to and removed from an activation list. The values of variables are stored in the state vector. Therefore it seems that the data structures involved during transition execution are: channel queues, an activation list containing activation records and the state vector.

The state vector must be kept as small as possible to minimise the size of the bit vector: n bits in the state vector require a bit vector with 2^n entries. A small state vector thus implies efficient memory usage during model checking. To reduce the size of the state vector a compact representation of the contents of data variables is defined in section 5.1.3. This compaction technique allows the minimum number of bits to be allocated to each variable in the state vector.

5.1.1 Communication Channels

In ESML an input (output) instruction is executable if and only if a matching output (input) instruction can be executed simultaneously—input and output instructions are said to synchronise. Input and output instructions are translated into functionally equivalent input and output transitions. Because transitions are executed one at a time, an input-output pair of transitions must be executed one before the other. A method to synchronise input and output transitions is needed.

Initially the approach adopted in the run-time environment of Joyce [9] was implemented. A message queue is allocated for each channel declared in a model. When an input (output) is executed and a matching message is not found in the appropriate channel queue a message

is appended to the channel queue and the current ESM is *blocked*. When an output (input) is executed and a matching input (output) has already occurred the relevant message is removed from the channel queue and the sending (receiving) ESM is *unblocked*. A guard in a POLL is executed by searching the channel queue for a matching communication and if none is found the next guard is executed. A message record is not added when a communication guard cannot execute because if one of the other guards matches, all the queue updates made by the previous POLL guards would have to be reversed. The ESM containing the POLL is only blocked if none of the POLL's guards can be matched.

Unfortunately this channel queue approach has the disadvantage that two communication guards in different POLLs cannot match. In the language definition of Joyce [10] it is stated that the guards of a POLL may not be matched, but in ESML this is allowed. After some unsuccessful attempts to extend the channel queue approach to cater for this scenario it was decided to implement synchronous communication without using channel queues[1]. This approach has the advantage that it also reduces the memory requirements.

When an output (input) is executed it *examines* all the ESMs in the model to see if a matching input (output) can execute, and if possible, completes the synchronisation. If no matching communication transition is executable the current ESM is blocked. If the communication is however in the guard of a POLL the ESM is only blocked when none of the guards (communication transitions) in the POLL can be matched. Examining the ESMs to find a matching communication entails checking whether the next transition to execute in an ESM is a matching communication. If the next transition is a guard of a POLL all the guards are examined. Holzmann use a similar technique for implementing synchronous communication for Promela [51]: when an output is executed it adds a message to a channel queue (the queue has only one slot) and then checks whether a matching receive is possible; if not, the message is removed from the queue.

Channel variables never hold any values because communication is unbuffered and synchronous. Consequently channel variables require no space in the state vector which reduces its size. This optimisation is not possible when asynchronous communication is supported as is the case with the validation language Promela [51].

[1]I would like to thank Hans Loedolff who implemented this communication method.

5.1.2 Activation Records

The memory allocation is inspired by the well-known method for implementing recursive, sequential procedures: it uses dynamic allocation of fixed size memory segments (*activation records*) which are not relocated during their existence. When a procedure is activated space is allocated in its activation record to store the values of its data variables during execution. In the run-time environment for ESML an activation record is allocated for each ESM activated in a model. Collectively, the data variables (except channel variables) and the location variable of each ESM are used to define the state of a model. The reachability graph of states generated by a model is used to check whether a given CTL formula is satisfied or not. Therefore it was decided to allocate the data and location variables storage within the state vector rather than in the activation records. Since the complete system state is stored in the state vector, manipulating the state during model checking is easier.

For a concurrent program a tree of activation records is a natural memory structure. In each branch of the tree the activation records are added and removed in last in first out order. However, allocating a tree of activation records in sequential memory presents a problem. In the implementation of the Joyce run-time environment [9] Brinch Hansen adopted the following approach: memory is organised as a single last in first out list of activation records. Hence the activation record of a terminated ESM is only removed when it is at the end of the activation list. Since ESML models represent reactive systems, ESMs seldom terminate. Therefore a last in first out *activation list* is implemented to organise activation records during the execution of ESML models.

Initially the activation list contains only one activation record: that of the initial ESM. When an ESM is activated an activation record is appended to the back of the activation list and the ESM is said to be in the *running* state. Each activation record is uniquely identified by its *activation number* which is determined by its position in the activation list. For brevity the ESM activation number is also referred to as the ESM number. An ESM enters the *terminated* state when it executes its termination transition. Since a terminated ESM's activation record is only removed when it is at the end of the list it will not be removed before all the ESMs activated by it have terminated.

When an ESM is activated part of the state vector is allocated to store its variables as

shown in Figure 9. All the bits allocated to an ESM are set to 0 to indicate the initial values of all the variables are zero. The part of the state vector allocated to an ESM which has terminated is not reused.

Each channel variable declared in an ESM is assigned a unique number when an ESM is activated. This *channel number* is used to identify communications via the channel. During compile-time each symbol class in a channel alphabet is also assigned a unique number which is used to determine if a message is of the correct symbol class during communication. If a channel variable is the i-th channel variable declared in ESM n then its channel number is $m * n + i$ with m the maximum number of channels that an ESM may declare. The values m (system constant) and i are known during compile-time and n is assigned during ESM activation. Therefore channel numbers need not be stored during run-time, because the formula can be used to uniquely identify the channel whenever it is referenced in the ESM. A port parameter uses the channel number of the channel variable (actual parameter) it corresponds to. In the example of Figure 9 the channel variable `User` has channel number $10 * 0 + 1$ if it is assumed that each ESM may only declare 10 channel variables. The port parameter `Semaphore` in both ESM `Proc` and ESM `Semaphore` therefore has the channel number 1 (see Figure 9).

Each activation record consists of the following fields:

Id
State
Parameters
Start

The **Id** field is used to uniquely identify all transitions belonging to the ESM. The **State** of the ESM can either be *running*, *blocked* or *terminated*. A pointer to the actual parameters of the ESM is stored in the **Parameters** field. The parameters are stored in a linked list with each element in the list either an expression value or a channel number. The **Start** field indicates the number of the first bit in the state vector allocated to variables in the ESM.

In Figure 9 an activation list in table format (growing downward) for a mutual exclusion model is shown after ESM `MutExcl` has terminated.

```
ESM MutExcl;
TYPE SemAlpha = {P,V}; Regions = NonCrit,Try,Crit;
VAR User : SemAlpha;

  ESM Proc(OUT Semaphore : SemAlpha);
  VAR Loc : Regions;
  BEGIN
    Loc := NonCrit;
    DO TRUE ->
      Loc := Try; Semaphore!P;
      Loc := Crit;
      Loc := NonCrit; Semaphore!V
    END
  END Proc;

  ESM Semaphore(IN Semaphore : SemAlpha);
  VAR Used : BOOLEAN;
  BEGIN
    Used := FALSE;
    DO TRUE ->
      POLL
        Semaphore?P/\~Used -> Used := TRUE
      []Semaphore?V        -> Used := FALSE
      END
    END
  END Semaphore;
BEGIN
  Proc(User);Proc(User);Semaphore(User)
END MutExcl;
```

State Vector : 19 Bits						
MutExcl	*Proc*		*Proc*		*Semaphore*	
Location	Data	Location	Data	Location	Data	Location
0..1	2..3	4..7	8..9	10..13	14	15..18

	Activation List			
ESM Number	Name	State	Parameter (Channel Number)	Start
0	MutExcl	Terminated	...	1
1	Proc	Running	1	2
2	Proc	Running	1	8
3	Semaphore	Running	1	14

Figure 9: The state vector and activation list for the mutual exclusion model. *After the initial ESM has terminated the state vector has a length of 19 bits and the activation list contains four activation records.*

5.1.3 Compact Variable Representation

In ESML a type defines a finite set of values. A variable of each type can hold one of these values at a time. To minimise memory requirements variables are stored in encoded form. For example if a subrange type `T = 0..255` is declared the contents of a variable of type `T` can be stored in 8 bits (1 byte).

For effective access only simple types are usually encoded. However, during model checking minimising the space required to store the contents of a variable is more important than the speed of the variable access. Therefore, the contents of *all variables* in an ESML model is encoded in $\lceil log_2 n \rceil$ bits, where n is the cardinality of the set representing the variable's type.

Boolean, Enumerated and Subrange Types

Since the standard type BOOLEAN is represented by the set { FALSE $\rightarrow$ 0, TRUE $\rightarrow$ 1} a BOOLEAN variable is encoded in 1 bit. Typically in a compiler for a programming language a byte is allocated to a boolean variable because a byte comparison is a fast operation to execute. Enumerated types are mapped onto the cardinals by mapping the first enumeration onto 0, the second onto 1, etc. As an example, type `T = red,green,blue` is mapped onto the set $\{0, 1, 2\}$ and a variable of type `T` is encoded in 2 bits. A subrange type is trivially mapped onto the cardinals. For example type $T = 0..5$ is mapped onto the set $\{0, 1, 2, 3, 4, 5\}$ and an element of the set is encoded in 3 bits.

BOOLEAN, enumerated and subrange types are encoded during the compilation phase and from then on only the encoded values are used. Therefore no encoding and decoding functions are required for variables of these simple types during the execution of transitions.

Record Types

For a record type the cartesian product of the sets representing the record fields are taken to produce the set representing the record type. This resulting set is then mapped onto the cardinals. Let s_i be the cardinality of the set representing field i of a record. Then a record variable with $n+1$ fields can be encoded in $\lceil log_2 \prod_{i=0}^{n} s_i \rceil$ bits. To encode the record

fields separately would in general require more bits as shown by the inequality

$$\lceil log_2 \prod_{i=0}^{n} s_i \rceil \leq \sum_{i=0}^{n} \lceil log_2 s_i \rceil$$

The mapping of a set representing a record type onto the cardinals is based on the *radix* notation to encode a number V in a specific base r:

$$V = d_0 r^0 + d_1 r^1 + d_2 r^2 + \ldots + d_n r^n$$

For records a *mixed radix* notation is used because each set representing a field type can have a different cardinality. Let d_i and s_i respectively be the value of the record field f_i and the cardinality of the set for the type of f_i. Then the encoded value V of a record with $n+1$ fields is given by:

$$V = d_0 r_0 + d_1 r_1 + d_2 r_2 + \ldots + d_n r_n$$

with $r_i = \prod_{j=i+1}^{n} s_j$

Consider the ESML record `T = (x,y:int)` with `int = 0..4`. The mapping constructed for this record is represented by the set: $\{(0,0) \rightarrow 0, (0,1) \rightarrow 1, (0,2) \rightarrow 2, (0,3) \rightarrow 3, (0,4) \rightarrow 4, (1,0) \rightarrow 5, \ldots, (4,3) \rightarrow 23, (4,4) \rightarrow 24\}$. A variable of this type is encoded in 5 bits. If the two fields were however encoded separately it would have required $3+3=6$ bits.

Encoding and decoding functions are required to manipulate the contents of record variables during the execution of transition code. If V is the encoded value of a record variable q, then the value of field i of q is given by

$$d_i = (V \text{ DIV } r_i) \text{ MOD } s_i$$

This function can be defined as **GetField**(V, r_i, s_i) with the values r_i and s_i known at compile-time.

When the value of field i is modified from d_i to d_i' the difference between the two encoded values is

$$\begin{aligned} V' - V &= d_0 r_0 + d_1 r_1 + \ldots + d_i' r_i \ldots + d_n r_n \\ &\quad -(d_0 r_0 + d_1 r_1 + \ldots + d_i r_i \ldots + d_n r_n) \\ &= (d_i' - d_i) r_i \end{aligned}$$

The new encoded value V' is thus given by $V' = V + (d'_i - d_i)r_i$. Changing a record field can be defined by the function **ChangeField**(V, r_i, s_i, d'_i) which firstly calculates the old value (d_i) of the field by using the **GetField** function and then returns a new encoded value V' for the record variable.

List Types

A list is a finite sequence. For a list type the set of all possible subsequences is computed. This resulting set is then mapped onto the cardinals. If the cardinality of the element set is s there are s^i subsequences of length i. Therefore the cardinality of the set representing the list type is:

$$s^0 + s^1 + s^2 + \cdots + s^n = \sum_{i=0}^{n} s^i = \frac{s^{n+1} - 1}{s - 1}$$

where n is the number of elements in the list and $s > 1$. A variable of a list type can therefore be encoded in $\lceil log_2 \frac{s^{n+1}-1}{s-1} \rceil$ bits.

Mapping the set representing a list type onto the cardinals require only a fixed radix because all the elements of a list have the same type. Let $x_i = \sum_{j=1}^{i-1} s^j$ be a function of the current length i of the list and d_j the value of element j in the list $(j \leq i)$. Then the encoded value of a variable of a list type with current length i is:

$$V = x_i + d_0 r^0 + d_1 r^1 + \ldots + d_{i-1} r^{i-1}$$

with r $(= s)$ the cardinality of the element set. In the rest of the discussion r is replaced by s. The encoded value of the empty list is the cardinality of the set representing the list type minus one. It was found to make no difference in the complexity of the encoding if the empty list is encoded as value 0. By using the cardinality of the element set s as a radix the mapping for the list `T = LIST[1] OF int` with `int = 0..1` is represented by the set: $\{ <0> \rightarrow 0,\ <1> \rightarrow 1,\ <0,0> \rightarrow 2,\ <1,0> \rightarrow 3,\ <0,1> \rightarrow 4,\ <1,1> \rightarrow 5,\ <empty> \rightarrow 6\}$. A variable of this list type is encoded in 3 bits.

In ESML there are four operations defined on a list variable: **LEN**, **HD**, **TL** and **::**.

The **LEN**(V,n) operation returns the length of a list for the current encoded value V and is defined as $i\ MOD\ (n+1)$ with i such that

$$(x_i \leq V < x_{i+1})$$

where n is the maximum number of slots in the list. The value n is known at compile-time.

If the encoded value of a variable of a list type is given by

$$V = x_i + \sum_{j=0}^{i-1} d_j s^j \tag{7}$$

it follows that

$$d_0 = V - \sum_{j=1}^{i-1} s^j - \sum_{j=1}^{i-1} d_j s^j$$

From this it can be seen that the first element (head) of the list can be retrieved with

$$d_0 = V \text{ MOD } s$$

since $d_i < s$ and $(\sum_{j=1}^{i-1} s^j + \sum_{j=1}^{i-1} d_j s^j) \text{ MOD } s = 0$. A function **HD**(V,s,e) with e the encoded value of the empty list returns the first element of a list if $V \neq e$. The encoded value of the empty list and the cardinality of the element set s are calculated during compilation.

The **TL** operation returns the encoded value V for the list without the first element:

$$V = x_{i-1} + \sum_{j=1}^{i-1} d_j s^{j-1}$$

This value can be interpreted as all the elements of the list shifted one left and the length of the list reduced by one. From equation 7 it follows that

$$\begin{aligned}(V - x_i) \text{ DIV } s &= \lfloor \frac{\sum_{j=0}^{i-1} d_j s^j}{s} \rfloor \\ &= \lfloor \frac{d_0}{s} + \sum_{j=1}^{i-1} d_j s^{j-1} \rfloor \\ &= \sum_{j=1}^{i-1} d_j s^{j-1}\end{aligned}$$

Therefore the **TL** operation performed on a list with encoded value V will result in the following encoding

$$V' = ((V - x_i) \text{ DIV } s) + x_{i-1}$$

This function is calculated by **TL**(V,s,e,n) if $V \neq e$. If $LEN(V, n) = 1$ the function **TL** returns the value e.

Appending an element d_i to the back of a list with length i and encoded value V will result in the new encoding

$$V' = \sum_{j=0}^{i-1} d_j s^j + d_i s^i + x_{i+1}$$

This encoding is interpreted as the old list with an element added at the back and the list length incremented by one[2]. Therefore appending an element to the back of a list with length i is defined as

$$V' = (V - x_i) + d_i s^i + x_{i+1}$$

This function is calculated by $\mathbf{Back}(V, d_i, s, e, n)$ if $LEN(V, n) + 1 \leq n$. If $V = e$ the function returns the value d_i.

Concatenating an element d_0 to the front of a list with length i results in the following encoding

$$V' = d_0 + \sum_{j=1}^{i} d'_j s^j + x_{i+1}$$

with $d'_j = ((V - x_i) \text{ DIV } s^{j-1}) \text{ MOD } s$ the values of the elements of the list before the concatenation operation. This encoded value represents a list with all its elements shifted one right, adding a first element and incrementing the list length by one. Concatenating an element to the front of a list is defined by the function $\mathbf{Front}(V, d_0, s, e, n)$ if $LEN(V, n) + 1 \leq n$. If $V = e$ the function returns the value d_0. Note that adding an element to the front of a list is a more complex operation than appending an element to the back of a list. For this reason if two lists are concatenated it is considered an append operation and the compiler will generate code to execute the **Back** function.

5.1.4 State Vector Access

The state of a system changes when the value of a variable in the state is changed. An implementation decision was taken to limit the cardinality of a type to what can be encoded in a *machine word*[3]. This limit makes the execution of variable accesses faster. It also has the side effect that ESML models are kept simple because the amount of detail introduced in the data part of the model is limited.

[2]If a list is concatenated to another then the length of the resulting list is the combined length of the two sub lists.

[3]For the current implementation this is 32 bits.

To access a variable in the state vector the position of the first bit and the number of bits (*bitlength*) needed to store the variable's contents are required. The *offset* of a variable is its position relative to the **Start** bit of an ESM which is stored in its activation record. For example in `ESM Proc` (see Figure 9) the variable `Loc` has bitlength 2 and offset 0. Both the offset and bitlength of a variable can be determined at compile-time. The offset plus **Start** is the position of the first bit allocated for a variable. For example `ESM Semaphore` (see Figure 9) has **Start** 14 and therefore the position of variable `Used` is 14 (its offset is 0). The position of the location variable of `ESM Semaphore` is $14 + 1$ and the bitlength of its location variable is 4.

The value of the variable stored from bit *start* and with *bitlength* bits can be retrieved from the state vector, *state*, by the function **GetValue**(state,start,bitlength). When a new value x is assigned to a variable, x will be assigned to the bits from *start* to *start+bitlength–1* in the state vector. This state change is accomplished by the procedure **SetVar**(state,start,bitlength,x). If the value x is too large to be stored in *bitlength* bits a range error is produced and execution is aborted.

As an example of the code generated for the manipulation of structured variables consider the following ESM.

```
ESM Show;
TYPE
   int  = 0..9;   (* cardinality = 10 *)
   list = LIST[4] OF int;  (* list with 5 slots; cardinality = 111111 *)
   record = (x,y : int); (* radix of x = 10; radix of y = 1; cardinality = 100 *)
VAR
  r : record;  (* offset 0; bitlength = 7 *)
  k : list;    (* offset 7; bitlength = 17 *)
BEGIN
   r.y := 2;
   k := <>;     (* Initialise list k as empty *)
   k := k::2;
   r.x := r.y + (HD(k) * LEN(k))
END Show;
```

The code generated for the assignment `r.y := 2` is the following:

```
   temp := ChangeField(GetValue(state,start+0,7),1,10,2);
   SetVar(state,start+0,7,temp);
```

Firstly, the encoded value of the record variable `r` is retrieved from the state vector, `state`, by executing `GetValue(state,start+0,7)` (`r` has bitlength 7 and offset 0). The value of

`start` is the value stored in the **Start** field in the activation record for `ESM Show`. If `ESM Show` is the only ESM in an ESML model `start` is 0. Calculating the new encoded value of `r` when field `y` is changed to value 2 is accomplished by the function `ChangeField(Value, radix_y, size_y, new)` with the parameters: `Value = GetValue(state,start+0,7)` , `radix_y = 1` , `size_y = 10` and `new = 2`. The new encoded value is stored in the variable `temp`. Lastly, the value of `temp` is assigned to the bits allocated to variable `r` in the state vector by executing `SetVar(state,start+0,7,temp)`.

The following code will be executed for the rest of the instructions in `ESM Show`:

```
(* k := <> *)
SetVar(state,start+7,17,111110);
(* k := k::2 *)
temp := Back(GetValue(state,start+10,17),2,10,111110,5);
SetVar(state,start+7,17,temp);
(* r.x := r.y + (HD(k) * LEN(k)) *)
temp := GetField(GetValue(state,start+0,7),1,10) +
        (HD(GetValue(state,start+10,17),10,111110) *
         LEN(GetValue(state,start+10,17),5));
temp := ChangeField(GetValue(state,start+0,7),10,10,temp)
SetVar(state,start+0,7,temp);
```

The state vector for `ESM Show` contains 27 bits (7 for variable `r`, 17 for variable `k` and 3 for the location variable). If no compaction is used and it is assumed that each integer value is allocated 8 bits `ESM Show` will require a 64 bit state vector ($2 * 8$ bits for `r`, $5 * 8$ bits for `k` and 8 bits for the location variable). The compacted version will require a bit vector with 2^{27} entries whereas the uncompacted version will require 2^{64} bits in the bit vector.

If no variable compaction is used the state vector size can be reduced by assuming a fixed length state vector and mapping the complete state onto this reduced state. Holzmann uses this scheme in the SPIN tool where a fixed state vector of 28 bits is assumed and each system state is hashed to obtain the index into a bit vector with 2^{28} bits. However, a hash conflict may cause errors in the model to be missed. Although the compaction techniques described here has a speed penalty compared to compaction by hashing it cannot effect the results obtained by the model checker. On average it was found that SPIN executes 30% faster than our model checker.

5.2 State Generation

The model checker determines whether a CTL formula is satisfied in the start state of a transition system. If the truth value of the formula cannot immediately be determined new states are generated by executing transitions in the transition system. This section explains the structure and execution of the state generation part of the model checker.

The state generator is defined by the procedure **NextState** which takes as input the current state, current ESM and transition and generates a new state by executing the transition. If no transition is executable in the current ESM another running ESM is scheduled for execution. The structure of the **NextState** procedure is discussed in section 5.2.3.

5.2.1 ESM Scheduling

ESMs execute concurrently, but when generating all possible execution paths of a model the state generator must generate all the possible interleavings of transitions in the different ESMs. The execution of an ESML model is therefore similar to that of a concurrent system being executed on a single processor. An algorithm for scheduling ESMs is required.

The scheduling of ESMs is done according to the ESM state stored in the ESM's activation record. Only running ESMs can execute transitions. The order in which ESMs are activated is used to schedule ESMs for execution. The activation order is determined by the activation numbers of the ESMs: if ESM A has a lower activation number than ESM B, A was activated before B. In section 5.1.2 it is explained that the activation number of an ESM indicates its activation record's position in the activation list. Therefore the scheduler searches the activation list, from a certain activation record onwards until the end, for the first activation record with a running state.

In the run-time environment of both Joyce and Promela a queue of running processes (*ready queue*) is used for scheduling. When an ESM terminates or is blocked it is removed from the ready queue. Because ESML models in general contain only a few active ESMs it is acceptable to schedule ESMs by examining the activation list.

If no running ESM can be scheduled and the current state s is a leaf state in the reachability graph—no successor state of s exists—a deadlock has occurred. When a deadlock occurs

the state generation is halted with a run-time error. Although the termination of all ESMs constitutes a deadlock, it is considered an *acceptable* state and model checking will continue.

5.2.2 Generating All Execution Paths

The difference between the execution of code for a programming language and the code for a validation model is, that for a program only one execution path is generated while for a model all the execution paths must be generated. The state generator must therefore be able to generate all execution paths for a transition system generated from an ESML model. Therefore in each state all the enabled transitions must be executed.

In Chapter 2 it is explained that a stack is used to store the current execution path during a depth-first generation of the reachability graph for a transition system. Stack entries are pushed when new states are generated and removed (popped) when the truth value of a CTL formula has been determined in the current state. The stack entries are also used to store information required to execute other execution paths when the entry becomes top-of-stack. Each stack entry contains the following fields:

state vector — The current state of the system.

activation list — The activation list together with the state vector define the execution environment for transitions and is therefore stored in the stack entry.

ESM Number — When the stack entry becomes top-of-stack this ESM will execute.

Transition Number — Indicating the transition that will be executed in the ESM (see field above) when this entry becomes top-of-stack.

When a stack entry is removed the state and the activation list in the new top-of-stack entry respectively become the current state and activation list used during state generation.

There are three reasons why more than one transition can be executable in a state:

1. More than one guard of a nondeterministic control structure (DO, IF or POLL) are true.

2. Transitions may be executable in more than one of the currently running ESMs in the model.

3. A communication transition can synchronise with more than one matching communication transition. For example, in the mutual exclusion model of Figure 9 if both processes request the semaphore the input transition for `Semaphore?V` can be matched by either of the two output transitions for `Semaphore!V`.

As an example of the use of the stack to generate all execution paths from a state consider the following nondeterministic IF and its corresponding transitions. The other two cases where more than one execution path must be generated are handled in a similar fashion.

```
x := 5;
IF x >= 5 -> z := x
[] x <= 5 -> z := x+5
END;
```

Number	NextTrans	Code	NextGuard
0	1	$x := 5$	...
1	2	$x >= 5$	3
2	6	$z := x$	...
3	4	$x <= 5$	5
4	6	$z := x + 5$	...
5	...	...	IF

After executing transition 0 to generate state s' a stack entry for s' is pushed onto the stack. After transition 1 has been executed, in state s', the top-of-stack entry is changed to indicate that whenever s' becomes the current state again transition 3 must execute. This ensures that from the state s' two execution paths are generated: firstly, the one where transition 1 (the first guard of the IF) is executed and secondly transition 3 (which corresponds to the second guard of the IF).

5.2.3 The NextState Procedure

The correctness of the model checker relies on the correctness of the state generator. Therefore it must be shown that the state generator will respond to every condition (situation)

that may occur during transition execution. A case analysis of the conditions influencing state generation will be given. It is shown that the state generator is complete in the sense that all execution conditions have been considered. The tasks performed in each situation will be outlined.

After executing a transition in the current state s the following four conditions influence the behaviour of the state generator:

NewStateGenerated — Was a new state, say s', generated during the execution of the transition? This condition is true if a communication, ESM activation, assignment, SKIP or expression guard has executed. The state s' must be returned to the CTL evaluator to allow the model checker to continue.

IO (Input-Output) — This is true if either an input or output transition has executed. Even if no matching communication can be performed, this condition will be true.

Guard — Does the transition correspond to the guard of an ESML control structure? Note that this guard transition may have failed.

Termination — This condition is true if an ESM termination transition has executed.

Control — This condition is true if the next transition to execute is a control transition. A control transition is executed when all the guards of a control structure (DO, IF or POLL) are false in the current state.

The conjunction of these 5 conditions results in 32 possible cases to be considered. This number can be reduced if it is observed that some of these conjunctions cannot occur.

1. A transition cannot simultaneously be a communication and a termination transition, a guard and a termination or a control and a termination transition. Therefore the conjunction **Termination** $\wedge$ (**IO** $\vee$ **Guard** $\vee$ **Control**) is false.

2. An ESM termination does not change the values in the state vector and thus the conjunction **NewStateGenerated** $\wedge$ **Termination** is false.

3. A control transition is executed when all the guards of a control structure are false. Therefore the conjunction **NewStateGenerated** $\wedge$ **Control** is false. A control

transition is considered the last guard in a control structure; therefore the conjunction **Control** $\wedge\neg$ **Guard** is false.

4. If a new state is not generated ($\neg$ **NewStateGenerated**) then either a communication was unsuccessful, a guard was false or a termination or control transition executed. This implies the conjunction $\neg$ **NewStateGenerated** $\wedge\neg$ **IO** $\wedge\neg$ **Guard** $\wedge\neg$ **Termination** $\wedge\neg$ **Control** is false.

By using this information the 32 cases are reduced to the following 9 (given in an ESML IF construct):

IF	NewStateGenerated	$\wedge$	IO	$\wedge$	Guard	$\wedge$	$\neg$ Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_0
[]	NewStateGenerated	$\wedge$	IO	$\wedge$	$\neg$ Guard	$\wedge$	$\neg$ Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_1
[]	NewStateGenerated	$\wedge$	$\neg$ IO	$\wedge$	Guard	$\wedge$	$\neg$ Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_2
[]	NewStateGenerated	$\wedge$	$\neg$ IO	$\wedge$	$\neg$ Guard	$\wedge$	$\neg$ Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_3
[]	$\neg$ NewStateGenerated	$\wedge$	IO	$\wedge$	Guard	$\wedge$	$\neg$ Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_4
[]	$\neg$ NewStateGenerated	$\wedge$	IO	$\wedge$	$\neg$ Guard	$\wedge$	$\neg$ Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_5
[]	$\neg$ NewStateGenerated	$\wedge$	$\neg$ IO	$\wedge$	Guard	$\wedge$	$\neg$ Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_6
[]	$\neg$ NewStateGenerated	$\wedge$	$\neg$ IO	$\wedge$	Guard	$\wedge$	$\neg$ Termination	$\wedge$	Control	$\rightarrow$	S_7
[]	$\neg$ NewStateGenerated	$\wedge$	$\neg$ IO	$\wedge$	$\neg$ Guard	$\wedge$	Termination	$\wedge$	$\neg$ Control	$\rightarrow$	S_8
END											

Each S_i represents the action taken by the state generator when the guard i is true. Not all the actions in this specification are unique. In the following, guards which have identical actions are grouped together:

1. The guards of S_0 and S_1 denote the same situation. A state is generated by means of a successful communication. The top-of-stack is updated to indicate that another matching communication may be executable. Because a new state is generated it must be returned to the CTL evaluator.

2. The guard of S_2 denotes a unique situation. A new state is generated by the guard of either a DO or IF construct. All the execution paths must be generated for the DO or IF; therefore the next guard transition is entered into the top-of-stack entry. This ensures that when s is the current state again the next guard in the DO or IF will be executed. The new state s' is returned to the CTL evaluator.

3. The guard of S_3 denotes a unique situation. A new state is generated by an assignment, ESM activation or a SKIP. Therefore the top-of-stack is changed to indicate the next running ESM in the activation list as an alternative ESM to execute when state s becomes the current state again. This ensures that all the running ESMs will execute in state s. State s' is returned to the CTL evaluator.

4. The guards of S_4 and S_6 denote the same situation. A guard transition failed in state s. The transition for the next guard must be executed.

5. The guard of S_5 denotes the situation where a communication, that is not part of the guard of a POLL, failed. The guard of S_8 denotes the situation where an ESM has terminated. In both these cases the next running ESM must be scheduled for execution. If no ESM can be scheduled and a deadlock does not exist the current state s is returned to the CTL evaluator because all the execution paths leading from s have been traversed.

6. The guard of S_7 denote the situation where a control transition must be executed. If the control structure being executed is an IF a run-time error is generated because all the IF's guards may not be false. For a DO structure execution continues with the first transition after the DO. For a POLL structure the current ESM is blocked and a new ESM is scheduled in search of a matching communication transition to execute. If no ESM can be scheduled a communication deadlock has occurred and a run-time error is generated.

A pseudo code outline of the state generation procedure **NextState** is given below (`A0` $= S_0 = S_1$, `A1` $= S_2$, `A2` $= S_3$, `A3` $= S_4 = S_6$, `A4` $= S_5 = S_8$ and `A5` $= S_7$):

```
PROCEDURE NextState(VAR s : StateVector; ESM, Transition : CARDINAL);
BEGIN
  LOOP
    Execute a transition
    (* A0, A1 and A2 returns a new state s by exiting the LOOP *)
    IF NewStateGenerated AND IO THEN A0 END
    ELSIF NewStateGenerated AND Guard THEN A1 END
    ELSIF NewStateGenerated THEN A2 END
    (* A3, A4, A5 finds a new transition to execute *)
    ELSIF (NOT NewStateGenerated) AND Guard AND (NOT Control) THEN A3 END
    ELSIF (NOT NewStateGenerated) AND IO THEN A4 END
    ELSIF (NOT NewStateGenerated) AND Termination THEN A4 END
    ELSIF (NOT NewStateGenerated) AND Control THEN A5 END
  END
END NextState;
```

The **NextState** procedure was originally coded without doing a case analysis. When the procedure was analysed in the above fashion a number of programming errors were found.

5.3 CTL Formula Evaluation

When the CTL evaluator prompts the state generator for a new state it must check whether the state returned is indeed a unique state. If the state s was given as input to the state generator and state s' is returned there are four cases to consider:

Equal(s,s') — The state generator was unable to generate a successor state for s. This indicates that all the paths from s have been traversed in the reachability graph. The top-of-stack entry is removed and the state at the new top-of-stack becomes the current state s and all paths leading from s will then be traversed.

Stacked(s') — The state s' is found on the stack and indicates a loop in the execution path. This is required to handle fairness. The next path from s will be traversed.

Visited(s') — The state has been visited before but is not in the current path. In this case the next path from s will be traversed.

Unique(s') — The state s' has never been visited before and a stack entry is pushed for this state. The CTL formula is evaluated in s' and depending on the result new states may be generated.

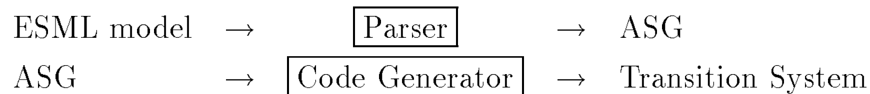
When a unique state s is visited the evaluator checks whether a CTL formula f is satisfied in the state. This is done by examining f according to the semantics of the CTL modalities in f. If at some point in this evaluation the value of a proposition, which forms part of f, is required the evaluator will execute the code for that proposition. If the CTL evaluator requires a new state to be generated from s the first running ESM in the activation list is scheduled for execution and **NextState** is executed. This ensures that all the currently running ESMs will have a chance to execute in state s.

In the next chapter the compiler generating the transition system and the transition code from an ESML model will be described.

Chapter 6

The ESML Compiler

Models written in ESML are translated into functionally equivalent transition systems by means of a three-pass compiler. The scanner and parser form the first pass and the code generator the second. A third pass (discussed in Chapter 7) was added to improve the efficiency of the transition systems generated by the compiler. The output of the parser is an abstract syntax graph (ASG) which is used by the code generation phase to generate the transition system. The first two passes of the compiler with their respective input and output represent two translation phases:



In this chapter the parser and code generator will be introduced. Results obtained during the compilation and model checking of ESML models are given in section 6.3.

6.1 Parser

The correctness of the compiler is more important than the speed of compilation. Standard techniques were used to implement the scanner [2, 8]. A recursive descent parser [8] was chosen for the parsing phase of the compiler. It was decided to exclude error recovery to

minimise potential errors in the compiler. ESML models are generally small and therefore error recovery is not essential.

6.1.1 Symbol Table

The symbol table entries are used for semantic checking and generating parts of the abstract syntax graph. Semantic checking is done in a straightforward way and is not discussed. The information required to generate the transition code for each ESML instruction is stored in the symbol table.

Normally when implementing a symbol table for a procedural language the symbol entries for a procedure are removed after a procedure has been parsed. The ESML syntax requires that the ESML model be parsed before the CTL formula. Since variables may be used in the model as well as in the CTL formula the symbol table must be kept in memory until the end of the parsing phase. For this reason the symbol table is implemented as a tree of linked lists with each level in the tree corresponding to a scope level in the ESM definitions (see Figure 10). Each linked list contains symbol entries for the constant, type, variable and ESM definitions contained in a specific ESM. Although the search time for a linked list is linear in the number of entries, each ESM only contains a small number of symbol entries.

The symbol table in Figure 10 is produced by parsing the accompanying ESML code. Each node in the tree contains the name of the ESM, a linked list of ESMs declared within the ESM and the linked list of symbol entries for the ESM.

The root of the tree contains an entry for the standard type BOOLEAN, the two boolean constants TRUE and FALSE, and the name of the main ESM in the ESML code (`ESM L0`). When an ESM is defined the entry for its name is made on the previous level to ensure that recursive activation is possible. For example `ESM L1a` is defined at level 2 but the entry for its name is made at level 1. Only local variables are allowed in ESML. Therefore a variable lookup requires only the local list to be searched. On the other hand a type, constant and ESM name lookup will require the local list to be searched as well as all the lists above it in the current branch of the tree. These two lookup procedures reflect the scope rules employed in ESML.

```
ESM L0;
TYPE t = 0..10;
VAR x : t;

   ESM L1a;
   VAR y : t;
   BEGIN ... END L1a;

   ESM L1b;
   TYPE t2 = 0..5;

      ESM L2a;
      VAR x : t2;
      BEGIN ... END L2a;

   BEGIN ... END L1b;

   ESM L1c;
   TYPE t = 0..20;
   VAR y : t;
   BEGIN ... END L1c;

BEGIN ... END L0;
```

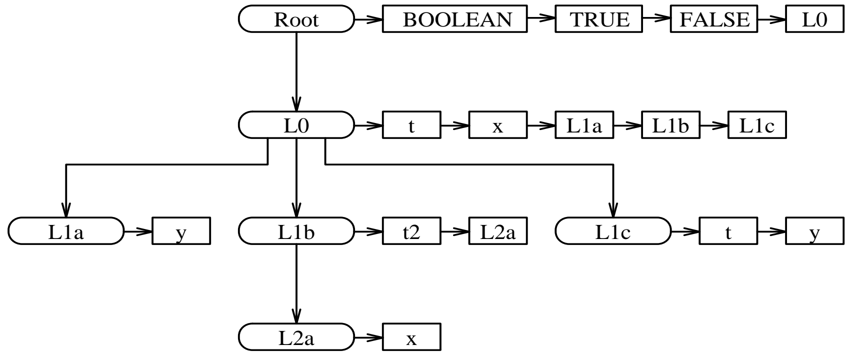


Figure 10: A symbol table (right) after the compilation of the ESML code (left)

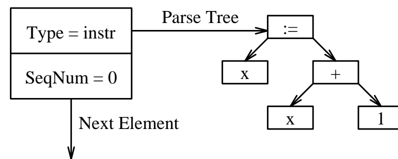
Each symbol table entry in an ESM is linked to the previous entry made in the ESM. This link is used for scope analysis. Each entry for a variable has a pointer to the entry for its type which is used for type analysis.

6.1.2 Abstract Syntax Graph

An abstract syntax graph is generated by the parser to allow optimisations to be performed before code generation. The abstract syntax graph is similar to the one used by Holzmann during the compilation of Promela models [51]. An ESM body is a sequence of ESML instructions. Therefore, the abstract syntax graph consists of a sequence of *elements* with each element corresponding to an ESML instruction. The elements in a sequence for a specific ESM are linked via a *next element* pointer to indicate the structure of a model.

Each element in a sequence has a *type* associated with it, indicating the kind of ESML instruction it represents. Assignments, communication, SKIP, ESM activation and expressions are of the *instr* type. A parse tree representing the ESML instruction is attached to each element of type *instr*. This parse tree is traversed during code generation to generate

the code to be executed for a transition. During parsing each element of type *instr* is assigned a unique *sequence number*. This number is used during code generation to generate the control flow information in the transition system. As an example of an element of type *instr* the element for the assignment `x := x + 1` is given below:



An element representing an ESML control instruction has type *IF*, *DO* or *POLL*. These elements contain only a sub-sequence field describing the guarded commands (options) in the control structure. Each option in the sub-sequence contains an element for the guard expression and another sequence of elements for the instructions associated with the action part of the command. Consider the following IF instruction with its corresponding syntax graph shown in Figure 11.

```
IF x > 0 -> x := x - 1
[] x = 0 -> x := x + 5
END;
```

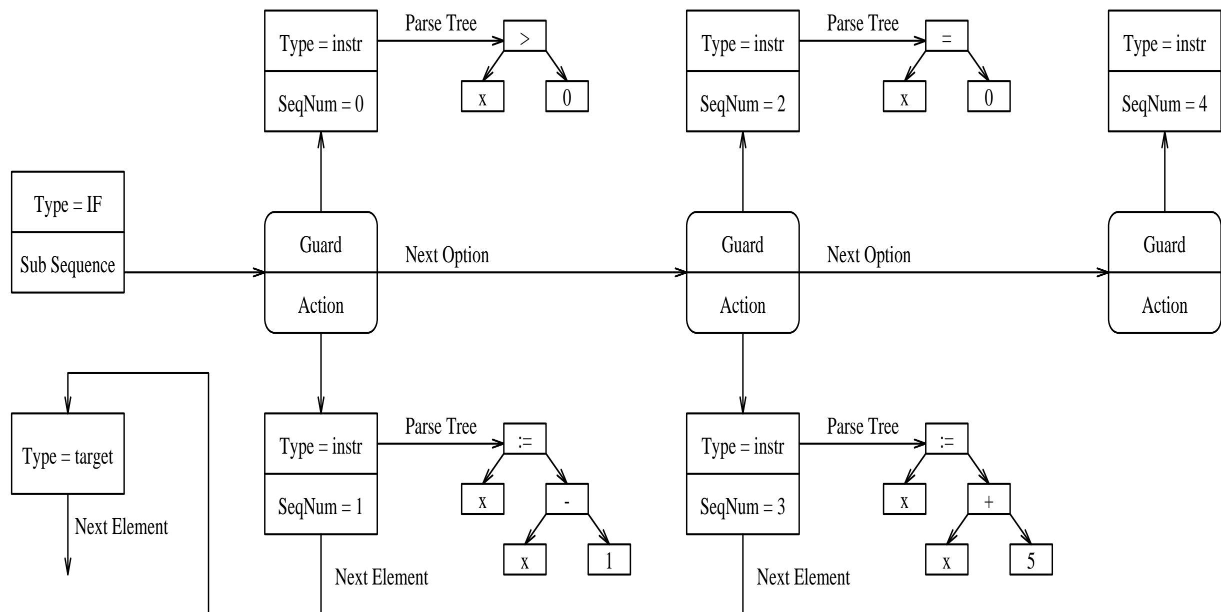


Figure 11: An abstract syntax graph for an IF construct.

An element of type *target* is used to indicate where the execution continues after a guarded command has executed. Since an IF and POLL terminate after the execution of one of

their guards a target element is added after the IF or POLL element in the sequence (see Figure 11). Since a DO is a repetitive construct its target element is added in the sequence before the DO element (see Figure 12). A similar technique is used to indicate control flow during the parsing of Promela models [51].

During parsing an option is added to the end of the sub-sequence of options to indicate that all the guards of a control structure can be false. This option has a guard element of type *instr* but it has no action part (see Figure 11). The *control* transition described in section 4.1.2 is generated for the guard of this option.

6.2 Code Generation

Translating the syntax graph into an equivalent transition system requires the translation of every element of type *instr* to a transition. The *target, IF, DO* and *POLL* elements are not translated to transitions, because their purpose is to indicate the control flow. The translation is done in two phases. Firstly, the control flow of each ESM is translated into the transition system format described in section 4.1. Secondly, each ESML instruction is translated into transition code.

As explained in section 4.3 each transition table entry is generated by a call to the procedure `MakeTrans(Esm, Trans, NextTrans, Code, NextGuard)`. During code generation the syntax graph is traversed and for each element of type *instr* a call to `MakeTrans` is generated. As an example of how the code is generated consider the following fragment of ESML code with its corresponding abstract syntax graph as shown in Figure 12:

```
ESM Code;
TYPE
  int = 0..10;
VAR
  x,y : int;
BEGIN
  x := 10;
  DO x > 0 -> x := x - 1
  [] x = 0 -> x := 10
  END;
  y := 5
END Code;
ASSERT AG((Code.x=0) => AF(Code.x = 10))
```

Figure 12: Syntax graph for `ESM Code`.

The fields of the transition entry (see section 4.1) for the assignment `x := 10` are the following:

Esm : The ESM number assigned during parsing. If it is assumed that `ESM Code` is the only ESM in an ESML model then its number is 0.

Transition Number : The value of the sequence number for the element, namely 0.

Next Transition : The value of the sequence number of the first element with type *instr* following the current element in the graph. For a guarded option the guard element is visited before the sequence of elements of the action. Therefore in the graph of Figure 12 this is the sequence number of the first guard element of the DO, namely 1.

Code : This field is assigned the index into the CASE structure where the code to be executed for this transition is stored. This is the first element in the sequence for `ESM Code` and therefore this field is assigned the value 1.

Next Guard : This field is empty. If this element was the guard element of an option then this field would have been assigned the value of the sequence number of the next guard option in the sub-sequence.

The code `MakeTrans(0,0,1,1,empty)` is therefore generated for the first element in the sequence of Figure 12. For the rest of the elements in the graph the following code will be generated

```
MakeTrans(0,1,2,2,3);
MakeTrans(0,2,1,3,empty);
MakeTrans(0,3,4,4,5);
MakeTrans(0,4,1,5,empty);
MakeTrans(0,5,6,empty,DO);
MakeTrans(0,6,7,6,empty);
MakeTrans(0,7,empty,7,empty);
```

The parse tree of each element is traversed to generate code for the transition. This is done in a standard textbook manner. The Modula-2 procedure generated for the code to be executed for each of the transition entries above is the following:

```
PROCEDURE ESMCode(VAR state : StateVector;
                  Index : CARDINAL);
VAR start : CARDINAL;
    expr  : StateValue;
BEGIN
   start := Value of the Start field of ESM Code's activation record;
   CASE Index OF
   1: SetVar(state,start + 0,4,10); (* Offset_x      = 0; Bitlength_x      = 4 *)
      SetVar(state,start + 8,3,1)   (* Offset_locvar = 8; Bitlength_locvar = 3 *)
  |2: IF (GetValue(state,start + 0,4) > 0) THEN
        SetVar(state,start + 8,3,2)
      END
  |3: expr := GetValue(state,start + 0,4) - 1;
      SetVar(state,start + 0,4,expr);
      SetVar(state,start + 8,3,1)
  |4: IF (GetValue(state,start + 0,4) = 0) THEN
        SetVar(state,start + 8,3,4)
      END
  |5: SetVar(state,start + 0,4,10);
      SetVar(state,start + 8,3,1)
  |6: SetVar(state,start + 4,4,5); (* Offset_y = 4; Bitlength_y = 4 *)
      SetVar(state,start + 8,3,7)
  |7: Set the ESM state terminated in the activation record for ESM Code.
   END
END ESMCode;
```

During parsing a parse tree is generated for the given CTL formula. During code generation this parse tree is traversed and a set of calls to the procedure `MakeCTLTree(Operator, left, right)` is generated as well as a procedure containing the code to be executed for each proposition in the formula. For example the following set of calls and code is generated for the formula `AG((Code.x=0) => AF(Code.x = 10))`:

```
   MakeCTLTree(AG,1,empty);
   MakeCTLTree(Imply,2,3);
   MakeCTLTree(1,empty,empty);
   MakeCTLTree(AF,4,empty);
   MakeCTLTree(2,empty,empty);


PROCEDURE EvaluateProposition(state : StateVector;
                              Index : CARDINAL) : BOOLEAN
VAR start : CARDINAL;
BEGIN
   CASE Index OF
   1: FindStart(start,0); (* ESM Code is assigned the number 0 during parsing *)
      RETURN GetValue(state,start + 0,4) = 0
  |2: FindStart(start,0);
      RETURN GetValue(state,start + 0,4) = 10
   END
END EvaluateProposition;
```

When the value of a variable is required to evaluate a proposition the position of the variable in the state vector must first be determined. Since the offset of the variable from the start of the variables for an ESM is known during compile-time only the value of the **Start** field in the ESM's activation record is required. The procedure `FindStart(start,esm)` returns the **Start** value of the first activation record in the activation list that has an **Id** field with value `esm`.

Summary

The output of the ESML compiler is a Modula-2 module containing the code for generating the transition system and the CTL tree as well as the transition and proposition code that are executed for generating new states and determining truth values respectively. The module therefore consists of four parts:

1. A set of calls to the `MakeTrans` procedure for generating the transition table.
2. A set of calls to the `MakeCTLTree` procedure for the generation of the CTL tree.
3. A procedure for each ESM containing the CASE structure with the transition code for that ESM.
4. The `EvaluateProposition` procedure with the code for the propositions in the CTL formula.

This module is compiled by a Modula-2 compiler and the output linked to the model checker code to generate an executable model checker for the ESML model and CTL formula. When executed the model checker firstly initialises the transition table and CTL tree and then executes the ESM activation of the initial ESM in the model and by so doing generates the start state of the system. This start state is evaluated by the CTL evaluator and depending on the result the state generator **NextState** (see section 5.2.3) is invoked to generate a new state to evaluate.

6.3 Results

The validation system is currently installed on a Data General AViiON 4265 with 128 Mb memory. The following output is produced during the validation of deadlock freedom in the process scheduler (given in Appendix A).

```
$mc scheduler.mdl
 ESML Compiler: Revision 93.05.25
 Input File : scheduler.mdl
 Generating Transition System...

 Compiling Model Checker...

 ESML Model Checker: Revision 93.08.11
 Specification Satisfied
 Unique states visited = 2032
 States revisited in stack = 766
 States revisited in bit vector = 3011
 Number of bits used in state vector = 56
 Longest execution path (Max. stack depth) = 315
```

The following four ESML models were selected to evaluate the performance of the validation system: the process scheduler, an elevator operating in a building with either 3 or 4 floors, a communication bridge between two networks and lastly a part of the X-Windows system. The models[1] were compiled and the resulting transition system executed to generate the complete state space of each model. In Table 1 the comparative figures obtained are given. The following five categories of comparison were chosen:

[1]The validation system is used as part of a graduate course on model checking at the University of Stellenbosch. I would like to thank the '93 class for constructing the models of the elevator, communication bridge and the X-Windows system.

Transitions — The number of transitions generated by the ESML compiler from a given ESML model.

Compile Time — This is the combined time (in seconds) required to generate a Modula-2 module from an ESML model and to compile and link this module to the model checker code.

Bits — The number of bits required in the state vector to store variables used in the model.

States — The number of states in the complete reachability graph of a given ESML model.

Time — The time (in seconds) required to generate the complete state space of a model.

ESML Model	Transitions	Compile Time	Bits	States	Time
Process Scheduler	61	2.9	56	2032	6.3
Elevator (3)	184	6.0	79	121989	410.5
Communication Bridge	89	3.7	68	163207	625.9
Elevator (4)	184	6.0	90	2546801	10942.1
X-Windows	162	5.8	111	$> 5 \times 10^6$	> 7000

Table 1: Validation results

The results in Table 1 indicate that the time needed to compile an ESML model is small compared to the time required to generate the complete reachability graph of a model. The number of transitions in the different models are small compared to the number of states generated for each model. The number of transitions in a model has no bearing on the size of the state space of a model. For example the two elevator models have the same number of transitions although the state space of the 4-floor elevator model is 20 times as large as that of the 3-floor model.

The X-Windows model behaved especially badly: after 1 hour 56 minutes (7000 seconds) and about 5 million states the available memory of the machine (128 Mb) was exhausted and the state generation had to be aborted.

It can be argued that during model checking it is in general unnecessary to generate the complete reachability graph because a given CTL formula may be either validated or

invalidated at any stage during the graph generation. However, in the case of a safety specification $AG(p)$ the complete graph must be generated to determine whether p holds in every state. Hence in the worse case all the states in the reachability graph must be generated.

Methods are required to reduce the number of states visited during model checking. This is an active field of research [4, 43, 52] and the focus of the next chapter.

Chapter 7

Optimisations

From the validation results given in the previous chapter (Table 1) it is clear that a state explosion occurs during the validation of complex models. In this chapter model reduction techniques are described to counter this state explosion problem. Model reduction allows memory efficient model checking because fewer states need to be stored during the generation of a model's reachability graph. Model reduction also reduces the validation time if the run-time overhead for implementing the reduction is small.

7.1 Model Reduction

The transitions from a given *partial ordering of transitions* in a concurrent system can be executed by *interleaving* ("shuffling") them in every feasible way. Consider two concurrently executing ESMs with the transition sequences t_1t_2 in ESM A and $t'_1t'_2$ in ESM B. From the partial ordering defined on the transitions from these two ESMs the following six interleavings $t_1t_2t'_1t'_2$, $t'_1t'_2t_1t_2$, $t_1t'_1t_2t'_2$, $t_1t'_1t'_2t_2$, $t'_1t_1t_2t'_2$ and $t'_1t_1t'_2t_2$ are possible.

A classical depth-first search algorithm executes all interleavings of a given partial ordering of transitions to generate all possible sequences of states. From [39, 77, 84] it is known that most of the state explosion due to the modelling of concurrency by interleaving can be avoided. In this section two techniques for avoiding the execution of unnecessary interleavings of transitions will be described. The first technique (section 7.1.1) ensures that

a state generated by one interleaving will not be generated again by another interleaving. The second technique (section 7.1.2) uses the information contained in the given CTL formula to reduce the number of states generated. Both these techniques are implemented during run-time. A compile-time optimisation is given in section 7.1.3 that uses the given CTL formula to reduce the validation model.

7.1.1 Sleep Sets

For the ESML model given in Figure 13 let the three assignments in `ESM X` and `ESM Y` generate the transitions t_1, t_2, t_3 and t'_1, t'_2, t'_3 respectively. The reachability graph generated while model checking the given CTL formula for this ESML model is given in Figure 14. The root of the graph (state 1) is the state generated after activating both `ESM X` and `ESM Y`.

```
ESM Example;

TYPE int = 0..5;

ESM X;
VAR x : int;
BEGIN x := 1; x := 2; x := 3 END X;

ESM Y;
VAR y : int;
BEGIN y := 1; y := 2; y := 3 END Y;

BEGIN X;Y END Example;

ASSERT AF(Example.Y.y = 3)
```

Figure 13: An ESML model

From the graph in Figure 14 it is clear that certain states are generated several times during the execution of different interleavings of the same partial order of transitions. For instance, state 11 is generated twice and state 9 is generated six times on different interleavings. *Sleep sets* [39, 42] can be used to eliminate this extra work. Each state visited in a depth-first search has a sleep set associated with it. A sleep set is defined as a set of transitions which may not be executed in a state, although these transitions may be enabled (guards are true) in the state. By using sleep sets during state generation it will be shown that for every partial order of transitions only one interleaving will be

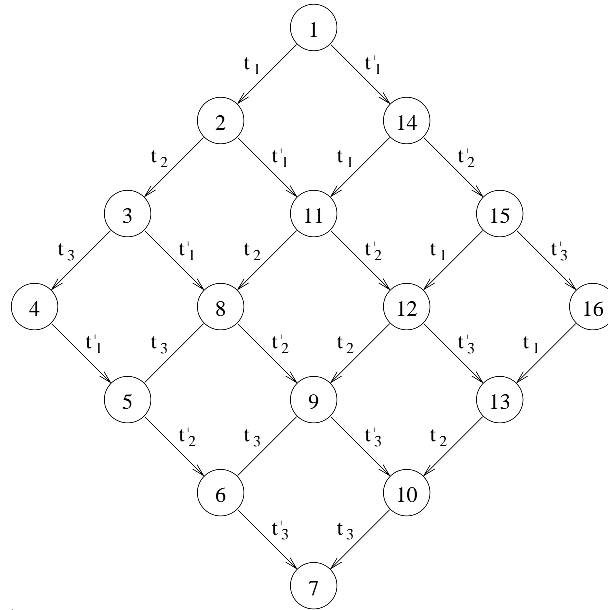


Figure 14: The reachability graph traversed during a depth-first search

fully executed. In other words the same state will not be generated by more than one interleaving of the same partial order of transitions.

The basic idea behind sleep sets is to avoid the execution of all possible interleavings of *independent* transitions. Dependencies arise between transitions if they refer to the same global variable or message channel. Since only transitions in different concurrent processes can be interleaved, only transitions from different processes can be dependent (or independent) on one another. The notation $t(s)$ is used to indicate the state generated when executing transition t in state s. Two transitions t and t' both enabled in state s are dependent if

1. t' is enabled in state $t(s)$ and t is enabled in state $t'(s)$, and
2. $t(t'(s)) \neq t'(t(s))$

In other words t and t' are dependent if executing the sequences tt' and $t't$ in a state s generate different resulting states; t and t' are independent if tt' and $t't$ generate the same state.

In state 1 of the reachability graph (Figure 14) both transitions t_1 and t'_1 are enabled. Assume that t_1 is executed before t'_1 and state 2 is generated. After all the enabled

transitions have been executed in state 2, t_1 is backtracked and state 1 becomes the current state. Transition t'_1 is now executed to generate state 14. Since t_1 and t'_1 are independent, t_1 is still enabled in state 14. Executing t_1 will result in state 11 being revisited because a previous interleaving of these two independent transitions, $t_1 t'_1$, has already generated state 11. To prevent execution of t_1 in state 14, t_1 is added to the sleep set of state 14.

In Figure 15 a recursive algorithm for a depth-first search with sleep sets is given. H is a cache where all the states visited in the search are stored together with their respective sleep sets. The set T indicates all the transitions enabled in the current state. The transitions in T are grouped according to the processes (ESMs) they form part of: $T = \{P_0, \ldots, P_n\}$ with the set P_i containing all the transitions enabled in process i. A synchronous communication is considered to be one enabled transition which forms part of either P_i or P_j if process i and j can communicate. Here we will assume an enabled communication transition belongs to the first process selected for execution in a state. The function `Order` performs the grouping of enabled transitions according to the processes they belong to. The function `Enabled` returns all the enabled transitions in a state s.

The algorithm shown in Figure 15 is essentially the one given by Godefroid and Wolper in [43]. The main difference is that transitions are grouped according to the process they belong to. Therefore all the enabled transitions in a specific process are executed in a state s before an enabled transition in another process is executed in s. The key elements of the algorithm are mentioned below.

1. Only transitions from different processes can be interleaved. Therefore when all the enabled transitions in a process have been backtracked to the current state s, these transitions are added to the sleep set at s.

2. In a state s all the transitions enabled in s but not in the sleep set at s may be executed and are stored in the ordered set T.

3. If a state is revisited, all the transitions must be executed that were in the sleep set when the state was first visited but is not in the current sleep set.

4. When a state s' is generated from s by executing transition t then all the transitions dependent on t are removed from the sleep set for state s'.

H is a cache where visited states together with their sleep sets are stored. The empty sleep set is indicated by {}. Since the sleep for the initial state s_0 is empty, (s_0,{}) is initially entered into H. The depth-first search is called with the initial state s_0, all the transitions enabled in s_0 and an empty sleep set: DFS(s_0,Order(Enabled(s_0)),{})

```
PROCEDURE DFS(s : state ; T : Ordered set of transitions; Sleep : SleepSet);
VAR
   s' : state;
   t : transition;
   Sleep' : SleepSet;
   T' : Ordered set of transitions;
   P,P' : Set of transitions enabled in a specific process;
BEGIN
   WHILE the set T is not empty DO
      (* Select and remove the set of enabled transitions of any process from T *)
      P := SelectProcess(T);
      P' := P;
      WHILE while there are still enabled transitions in the set P DO
         t := SelectTransition(P); (* Select and remove any transition t from the set P *)
         s' := t(s);
         IFs' ∈ H THEN (* s' has been visited before *)
           (* Deposit all the transitions that are stored in the sleep set of s' but
              are not in the current sleep set in the ordered set T'. *)
           T' := Order({t' | t' ∈ H(s').Sleep ∧ t' ∉ Sleep});
           (* Calculate the intersection of the transitions in the sleep set stored
              for s' and the current sleep set and assign this set to the entry for
              s' in the store H. *)
           Sleep' := H(s').Sleep ∩ Sleep;
           H(s').Sleep := Sleep'
         ELSE (* s' has not been visited before *)
           (* Remove all the transitions dependent on t from the sleep set *)
           Sleep' := Sleep \ Dependent(t);
           (* Store s' and its sleep set in the store *)
           Enter (s', Sleep') in H;
           (* Order the transitions enabled in s' but that are not in the sleep set
              according to the processes they belong to and assign this set to T' *)
           T' := Order(Enabled(s') \ Sleep')
         END;
         (* Execute all enabled transitions that are not in the sleep set in the state s' *)
         DFS(s',T',Sleep');
         (* Transition t is now backtracked and another enabled transition is selected
            for execution in state s *)
      END;
      (* All the transitions enabled in state s in the current process have now been executed.
         Add these transitions to the sleep set at state s *)
      Sleep := Sleep ∪ P'
   END
END DFS;
```

Figure 15: A depth-first search algorithm using sleep sets

Figure 16 shows the graph constructed by using sleep sets during the depth-first search of the ESML model given in Figure 13. All non-empty sleep sets are displayed in parentheses.

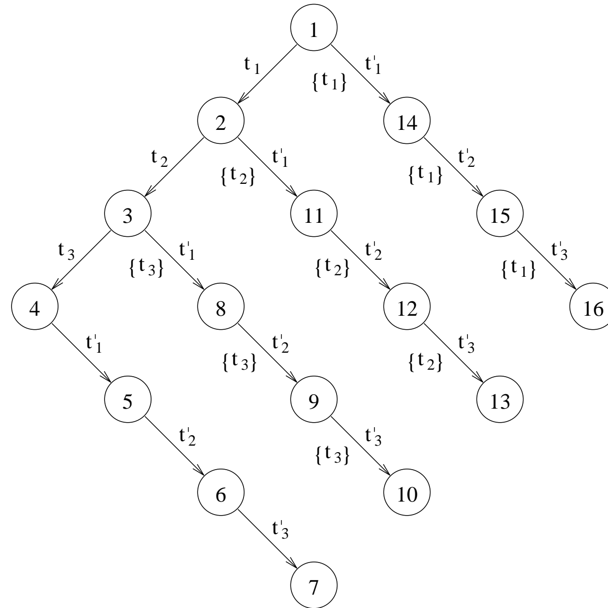


Figure 16: The reachability graph traversed during a depth-first search with sleep sets

In [43] Godefroid and Wolper proved a theorem stating that any state reachable from the initial state in which a deadlock occurs will also be reached when using a depth-first search with sleep sets. This proof could be adapted to obtain the same result for the algorithm given in Figure 15. This result allows the model checking of specifications that can be reduced to a state accessibility problem. Therefore, safety properties can be model checked by using this technique.

Liveness specifications can present a problem because executable transitions are ignored in certain states. As an example consider the ESML model of Figure 13 with a new liveness specification $AF(Example.X.x = 3 \wedge Example.Y.y = 3)$. This liveness specification is true if all paths from the start state can reach state 7 (see Figure 14). In Figure 14 the start state 1 will satisfy the formula but in Figure 16 state 1 will not satisfy the formula, because only the leftmost path can reach state 7. The following strategy was adopted to handle liveness specifications: when a state is visited from which no transitions are executable and the CTL formula is false in the state, but the current sleep set is not empty then the formula is considered true and the model checking continues. If the sleep set is not empty executable transitions exist that will generate already visited states when

executed. The depth-first execution of transitions ensures that these already visited states did satisfy the liveness specification, because if they violated the specification the model checking procedure would have already terminated. In Figure 16 with the liveness formula $AF(Example.X.x = 3$ /\ $Example.Y.y = 3)$ this strategy will ensure that the formula will be true in states $10, 13, 16$. Since these states are reachable from state 1 the formula is satisfied in the graph of Figure 16. Note that this strategy can also be used when precedence (section 2.1.2) and next-state (AX and EX) formulas are being used. This strategy is currently being used in the validation system and although it has been tested thoroughly, it has not yet been proven to be sufficient to ensure correct validation results.

It will now be shown that in ESML *no two transitions can be dependent.*

- In ESML a transition can only be enabled when the value of the location variable of the ESM it resides in points at the transition. Each transition is represented by a unique value of the location variable. Assume the transitions t and t' are enabled in a state s and t is enabled in state $s'' = t'(s)$ and t' is enabled in state $s' = t(s)$. For t and t' to be independent it must be shown that $t(s'') = t'(s')$. Transitions t and t' cannot be contained in the same ESM, because if we assume t executes first in s the value of the location variable changes and therefore t' is no longer enabled in state s'. Each ESM is allocated a part of the state vector to store the values of its variables. Since ESML models cannot contain global variables and t and t' are in different ESMs, t and t' change different parts of the state. The two sequences tt' and $t't$ will therefore generate the same state $(t(s'') = t'(s'))$.

The following assertion will now be shown to hold: *A depth-first search with sleep sets for an ESML model does not require the sleep set to be stored for every state visited during the search.*

- Since ESML models do not contain dependent transitions it follows that in the algorithm given in Figure 15 if a transition t is added to the sleep set it will never be removed again because no transition can be dependent on t. This means that if a state s is revisited the sleep set stored for s, namely $H(s).Sleep$, is always a subset of the current sleep set, $Sleep$. If a state s is revisited and $H(s).Sleep \subseteq Sleep$ there can be no transition t' such that $t' \in H(s').Sleep \wedge \; t' \notin Sleep$. Therefore from

the algorithm in Figure 15 no transition can be executed in state s when state s is revisited.

In Figure 17 the depth-first search with sleep sets is simplified for models that does not contain dependent transitions.

```
PROCEDURE DFS(s : state ; Sleep : SleepSet);
VAR
   s' : state;
   t : transition;
   T : Set of transitions grouped according to the process they belong to;
   P,P' : Set of transitions enabled in a specific process;
BEGIN
   T := Order(Enabled(s) \ Sleep);
   WHILE the set T is not empty DO
       P := SelectProcess(T);
       P' := P;
       WHILE there are enabled transitions in the set P DO
           t := SelectTransition(P);
           s' := t(s);
           IF s' ∉ H THEN
              Enter s' in H;
              DFS(s',Sleep)
           END
       END;
       Sleep := Sleep ∪ P'
   END
END DFS;
```

Figure 17: A depth-first search algorithm for ESML models using sleep sets.

In Figure 16 each state is only visited once. The following assertion can be shown to hold: *For every reachable state s, the depth-first search with sleep sets given in Figure 17, never completely explores more than one interleaving of a given partial order of transitions that leads to state s, and thus never visits state s twice because of the exploration of two interleavings.*

- Consider two interleavings of the same partial order of transitions: k and k'. Let the interleavings of k and k' be the same until state s. Let $Pref(k)$ denote the common prefix of k and k' that ends when k and k' differ after state s. Assume the next transition of k after $Pref(k)$ is t and that the next transition after $Pref(k')$ is

t'. Thus in state s both t and t' are enabled. Assume that t is first explored in s. Later, when t is backtracked in state s transition t is added into the sleep set. In k' transition t' is now executed but t cannot be executed again because it can never be removed from the sleep set. Thus it is impossible to generate a state in k' already visited by k.

State Space Caching

During state generation a *state space cache* [41] can be used to store the already visited states. The cache is of limited size and when it becomes full states are deleted to accommodate new states. When using a state cache, models with state spaces that are too large to be handled by the bit vector technique can also be validated. However, unnecessary work will be performed if a state that has been deleted from the cache is revisited later in the search. The success of this technique is unpredictable because it is dependent on the state space of the specific model for which states are being generated [41].

With the introduction of sleep sets only different partial orderings of transitions can generate the same state. Therefore already visited states will seldom be revisited. This allows state space caching to be used because a state that is deleted from the cache will now seldom be revisited. In [41] Godefroid *et al.* achieved promising results by implementing state space caching in conjunction with sleep sets. In the current implementation of the model checker (Chapter 2) the bit vector has been replaced by a cache. When the cache becomes full old states are over-written by the new states generated. Our experience is that when cache over-writes occur frequently run-time can go up dramatically.

Implementation

The sleep set technique was implemented in the run-time environment described in Chapter 5. However instead of storing transitions in the sleep set, ESM numbers (see section 5.1.2) are stored. This decision is justified because each ESM has a location variable value indicating the transition that must be executed in the ESM. When implementing a reduction technique during run-time the overhead incurred must not render the model checking procedure useless. The scheduler, elevator and the communication bridge (Bridge)

models (first mentioned in section 6.3) are used to compare a classic depth-first search with the search using sleep sets.

ESML Model	Search	Unique States	Visited States	Time (seconds)
Scheduler	DFS	2032	5809	6.3
	DFS+Sleep	2032	2686	2.8
Elevator(3 floors)	DFS	121989	369480	410.5
	DFS+Sleep	121989	152232	190.3
Bridge	DFS	163207	563314	625.9
	DFS+Sleep	163207	243860	325.1
Elevator(4 floors)	DFS	2546801	9300815	10942.1
	DFS+Sleep	2546801	3178251	3739.1

A state space cache with 6×10^6 entries requiring 96 Mb of memory was used to store the visited states. During the state generation of the X-Windows model the cache became full and, as a result, entries were over-written frequently. After more than a day the complete state space of the model was not yet generated and it was decided to abort the state generation. During the generation of the state spaces of the other models no cache over-writes occurred.

7.1.2 Partial Order Semantic Rules

In this section a partial order semantic rule is given that reduces the model according to the CTL formula to be validated. It uses the concept of *eligible* transitions to avoid executing certain unnecessary transitions in a state. A similar reduction rule based on eligible transitions is implemented in the run-time environment of Promela (SPIN) [52].

An eligible transition is a transition that changes the value of a global variable or refers to a message channel. Although ESML do not support global variables, the CTL formula is considered to be global because its value can be changed by a transition in more than one ESM. Therefore in ESML only communication transitions and assignment transitions that modify the value of a variable in the CTL formula are considered to be eligible. The partial order rule is the following:

- Whenever an eligible transition t such that $s \xrightarrow{t} s'$ is backtracked by removing the state s' from the stack all enabled transitions must be executed in state s.

The rule can also be interpreted as: whenever an ineligible transition t' such that $s \xrightarrow{t'} s'$ is backtracked by removing the state s' from the stack no enabled transitions in other ESMs need to be considered for execution in state s. The basic idea behind this rule is to avoid executing enabled transitions from other ESMs in a state s when these transitions have already been executed (during depth-first execution) in some state s' reachable from s. If two communication transitions t and t' are both enabled in the current state s it is possible that when t' is executed in s, t is disabled in the successor state s'. Therefore when a communication transition is executed in s all paths from s are generated. Since the reachability graph being generated is used for model checking, all the states that can affect the model checking result must be generated. Therefore when a transition changes the value of a variable in the CTL formula in state s all paths from s must be generated. This ensures that no state is missed that could change the validation result.

In the ESML model of Figure 13 only the transitions t'_1, t'_2, t'_3 are eligible, because they change a variable in the CTL formula. The transitions t_1, t_2, t_3 are ineligible and therefore t'_1 is not considered for execution in any of the states $1, 2, 3$. Figure 18 displays the reduced graph generated by using the partial order semantic rule for the ESML model of Figure 13.

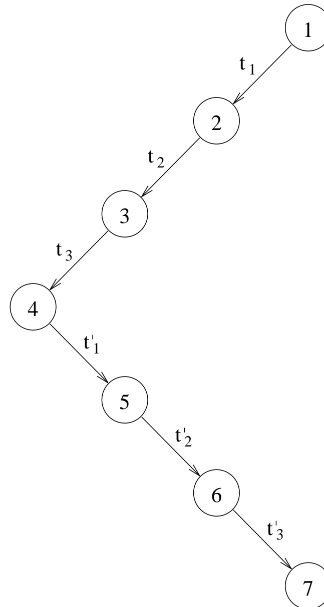


Figure 18: The reachability graph traversed guided by the partial order rule.

If the CTL formula was $AF(Example.X.x = 3 \wedge Example.Y.y = 3)$ then all the transitions $t_1, t_2, t_3, t'_1, t'_2, t'_3$ would have been eligible because all six transitions change the value

of a variable in the CTL formula. In this case no reduction in the graph of Figure 16 would have taken place because all the enabled transitions are executed in each state of the graph.

Determining which transitions are eligible can be done during the compilation of the ESML code and therefore requires no run-time overhead during model checking. The implementation of the partial order rule in the run-time environment of Chapter 5 is straightforward. Every state in the stack receives a flag to indicate whether the state can be skipped when backtracking. Therefore, if a state s' is generated from state s via an eligible transition the stack entry for state s is updated to indicate that it may not be skipped when backtracking. Note that the partial order rule has no effect on the generation of execution paths for a nondeterministic control structure; it is only used when different interleavings of transitions are concerned. If the next-state operators AX and EX are used in a CTL formula the partial order rule cannot be applied. These next-state operators do not necessarily have the same truth value for reachability graphs that are *equivalent with respect to stuttering* [12]. Fortunately, these two operators are easily evaluated because they are in the worse case only concerned with all the successor states of a state s.

7.1.3 Transition Folding

This technique allows a consecutive sequence of transitions in an ESM to be *folded* (grouped) into one transition. This optimisation is performed during the ESML compilation: when a sequence of consecutive transitions in an ESM are all ineligible they are sequentially grouped into one ineligible transition representing the transition sequence. Since ineligible transitions do not change the value of a variable in the CTL formula the states generated by executing these transitions can be removed without influencing the validation result. As is the case with the partial order semantic rule the next-state operators AX and EX cannot be used in conjunction with transition folding. Since the transitions from `ESM X` in the ESML model of Figure 13 are all ineligible, they can be folded into one transition T representing the sequence $t_1t_2t_3$. The execution of T represents the sequential execution of t_1 then t_2 and lastly t_3. In Figure 19 the graph from Figure 18 is further reduced by transition folding.

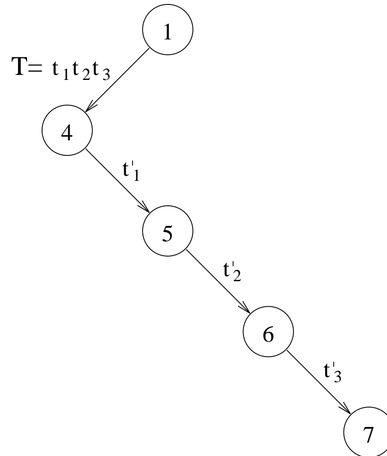


Figure 19: The reachability graph traversed with transition folding added.

7.1.4 Results

The following table illustrates the model reduction obtained by adding *sleep* sets, the *partial order semantic rule* (POSR) and *transition folding* (Folding) to the classical depth-first search (DFS) during the generation of the state space for the scheduler model given in Appendix A. For the partial order semantic rule and transition folding only communication transitions were considered eligible.

	DFS	Sleep	POSR	Folding	Sleep+POSR+Folding
States Explored	5809	2686	4450	4338	1112
Time (seconds)	6.3	2.8	4.9	4.6	1.4

In Table 2 the reduction in the number of unique states generated for the elevator, communication bridge and the X-Windows models are given. For the partial order semantic rule and transition folding only communication transitions were considered eligible. A state space cache with 6×10^6 entries requiring 96 Mb of memory was used to store the visited states. By adding the model reduction techniques it is possible to generate the complete state space of the X-Windows model. Due to a lack of memory this was previously impossible. For both the elevator models and the communication bridge model no cache over-writes occurred during state generation. While generating the state space of the X-Windows model over-writes did occur and therefore the number of unique states visited cannot be determined. The number of unique states in Table 2 for the X-Windows model is the number of states stored in the cache during state generation.

ESML Model	DFS		Sleep+POSR+Folding	
	States	Time	States	Time (seconds)
Elevator (3)	121989	200.1	43129	80.3
Communication Bridge	163207	254.7	46137	94.2
Elevator (4)	2546801	4244.7	901003	2011.4
X-Windows	$> 5 \times 10^6$	> 7000	4117623	7309.8

Table 2: Results obtained by adding model reduction techniques.

7.2 Future Work

Although the model reduction techniques described here reduce a model's state space, complex models can still be constructed that cannot be handled in the available memory. It may be possible to use compositional techniques [1, 16] to handle such models: instead of model checking a complete model, the model is decomposed into manageable parts which are validated separately. Another possibility to reduce the memory requirements during model checking is to use a binary decision diagram (BDD) [70] to represent the visited states instead of a state space cache or a bit vector. BDDs are used in a number of current model checking systems [34, 53, 71]. Other model reduction techniques which have been investigated include exploiting symmetry [22], abstractions [23] and unfoldings [72].

Chapter 8

Conclusions

The purpose of this thesis was to develop a compiler to translate high-level validation models, expressed in the language ESML, into functionally equivalent transition systems. This included the design of an efficient run-time environment for transition systems which forms the basis of several validation systems.

ESML was developed to support the validation of operating system kernels. Since the behaviour of such systems cannot be analysed without modelling complex data structures such as process queues, facilities such as lists and records were included in the language. A technique was developed to represent structured data as compactly as possible. A prototype of the validator has been completed and after using the system to analyse various non-trivial models, it is perhaps time to look back and evaluate what have been learnt.

8.1 Retrospective

The project gave me valuable insight into language design and compiler writing. At the time of writing the compiler and accompanying run-time environment, only a few validation systems existed and only the implementation detail of the SPIN system [51] was available to us. The main difference between a run-time environment for a typical programming language and that of a validation language is that a programming environment only requires one execution path to be generated by a program whereas a validation system may require

that all execution paths of a model must be generated. In the run-time environment of ESML a backtracking technique is used to execute all possible execution paths of a model.

We had the advantage that ESML could be designed with the explicit purpose of model checking in mind. It is important when designing a validation system to keep in mind the restrictions imposed by the validation technique adopted. A specification language that was not designed with model checking in mind (for example LOTOS [56] supports infinite data structures) will be much harder to use in conjunction with a model checker.

A philosophy adopted was to use tried and trusted techniques where possible. This led to the following implementation decisions:

- The **scanner** and **recursive descent parser** defined in [8] were adopted for the ESML compiler.
- The **abstract syntax tree** used in the ESML compiler is similar to that used in the Promela compiler (SPIN) [51].
- The structure of the **transition system** generated from the ESML model was adapted from a similar transition system used in the SPIN system.

Two exceptions to this philosophy are also worth mentioning: the compaction method and the partial order technique based on sleep sets. The compaction method was first defined and implemented and afterwards the mapping to the radix method of representing numbers of mixed base was found. This mapping illustrated the correctness of the functions defined to store and retrieve values forming part of records and lists. It also showed how arrays could be defined with the same notation as records by using a fixed radix. Although not in the language definition given in this thesis, an array structure has been added since. The partial order technique based on sleep sets was first enforced by the ESM scheduling function before the mapping to the work of Godefroid and Wolper [43] was made. This mapping illustrated the correctness of the sleep set implementation for ESML models.

Two main design decisions were the exclusion of global variables and adopting synchronous communication as the only means of message transfer between ESMs. These decisions simplified the implementation of the compiler and run-time environment. It might be argued that writing models of concurrent systems will be made more difficult by not allowing global

variables, but until now this is not reflected in our experience of using the ESML language. In fact, models seem to be more elegant (understandable) because of the modularity existing between ESMs. Above all a global variable can be modelled by using an ESM. The implementation of partial order techniques for reducing the reachable state space during model checking was also simplified by the absence of global variables and asynchronous communication.

From the experience gained by writing ESML models, the following enhancements are proposed to the ESML language:

- Functions are required because an ESM is not a natural representation of a function.
- Although ESML is object based full object orientation will make the language more flexible as well as more powerful.
- The state space of complex models can still be too large to handle. Therefore mechanisms are required to indicate that certain ESMs in a model must not be analysed although they can still interact with the ESMs in the rest of the model. This will allow compositional techniques to be used during validation.

8.2 Program Correctness

Computer systems are often used in safety-critical applications and it is becoming essential to prove the correctness of the software controlling such systems. In my opinion the only way this can be achieved is by rigorous mathematical prove of program correctness. Due to the inherently complex nature of concurrent programs constructing a prove of their correctness is difficult.

One possibility is to construct a model of the intended behaviour of a system and prove the model correct. If the model is too far removed from the real system proving that the system is a refinement of the model may be difficult. This brings us to a second possibility, namely to construct a model and generate the real system from this model. Since in general an implementation contains more detail than a model only a skeletal implementation can be generated automatically. Further detail will have to be included manually to develop a

full implementation. Potential errors in the system are thus localised to the coding of this detail.

Unfortunately, many programmers consider the construction of a model of a system, that can be proven to be correct, a waste of time. This may be due to a lack of confidence in the practicality and power of the validation techniques being used. The work presented here is an attempt to make model checking more usable by allowing the validation of models expressed in a high-level validation language that supports structured data.

Appendix A

Scheduler Model

A fixed number of processes is assumed. This is captured by the type definitions *procrecord* and *procqueue*. The model consists of four ESMs: *ready* which maintains the queue of process records, *running* which manages the currently active process, *user* which simulates the currently active user process and *device* which can respond to an IO request by generating an IO completion interrupt. ESMs *ready* and *device* act as servers to *running* and *running* in turn is a server to *user*. An alphabet of valid messages is defined for each channel: *readyrequest* for messages from *running* to *ready*, *readyresponse* for messages from *ready* to *running*, *trap* for both *device* to *running* and *user* to *running* and *iocommand* for messages from *running* to *device*.

The ESM *user* is an abstract model of user processes. A user process can nondeterministically decide to issue a kernel call or to enter a nonterminating loop.

A new process can only be selected when the process queue is nonempty. When a kernel call occurs it is assumed (for simplicity) that a user requested an IO operation. To start the IO operation, a copy of the process record of the requesting process is transferred to the IO device. When the IO completion interrupt occurs ESM *device* returns the process record of the process which requested the IO operation to *running*. To indicate that a process is waiting for an IO operation to complete, its state is set to *io*. Here is the behavioural specification:

```
ESM scheduler;

CONST maxproc     = 3;

TYPE
  procnumber    = 0..maxproc;
  procstate     = active, io;
  procrecord    = (number: procnumber;
                    state: procstate);
  readyrequest  = {createproc, selectproc(procrecord),
                   enterproc(procrecord)};
  readyresponse = {newproc(procrecord)};
  trap          = {kcall, interrupt(procrecord)};
  iocommand     = {doio(procrecord)};

VAR
  ch0 : readyrequest;  (*ready <= running*)
  ch1 : readyresponse; (*ready => running*)
  ch2 : trap;          (*device => running and user => running*)
  ch3 : iocommand;     (*running => device*)

  ESM ready(IN request : readyrequest; OUT response : readyresponse);
  TYPE
    procqueue = LIST[maxproc] OF procrecord;
  VAR
    proc        : procrecord;
    readyqueue  : procqueue;
    proccounter : procnumber;
  BEGIN
    readyqueue := <>;
    proccounter := 0;
    DO TRUE ->
      POLL
         request ? createproc ->
         IF proccounter = maxproc ->
            SKIP
         [] proccounter # maxproc ->
            proc.number := proccounter;
            proc.state := active;
            readyqueue := readyqueue::proc;
            proccounter := proccounter + 1
         END

      [] request ? selectproc(proc) /\ LEN( readyqueue ) # 0 ->
         IF proc.state = io ->
            SKIP (*throw away process*)
         [] proc.state = active ->
            readyqueue := readyqueue::proc
         END;
         response ! newproc(HD(readyqueue));
         readyqueue := TL(readyqueue)

      [] request ? enterproc(proc) ->
         IF proc.state = io ->
            SKIP (*throw away process*)
         [] proc.state = active ->
            readyqueue := readyqueue::proc
         END
      END (*poll*)
    END
  END ready;
```

```
  ESM running(OUT toready   : readyrequest; IN fromready : readyresponse;
               IN request   : trap;         OUT todevice : iocommand);
  VAR currentproc, proc : procrecord;
  BEGIN
    proc.state := io;
    toready ! createproc;
    toready ! createproc;
    toready ! selectproc(proc);
    fromready ? newproc(currentproc);
    DO TRUE ->
      request ? kcall;
      todevice ! doio(currentproc);
      currentproc.state := io;
      toready ! selectproc(currentproc);
      fromready ? newproc(currentproc)
    END
  END running;

  ESM device(IN fromrunning: iocommand; OUT toready: readyrequest);
  VAR proc: procrecord;
  BEGIN
    DO TRUE ->
      fromrunning ? doio(proc);
      proc.state := active;
      toready ! enterproc(proc)
    END
  END device;

  ESM user(OUT torunning: trap);
  VAR looping: BOOLEAN;
  BEGIN
    looping := FALSE;
    DO ~looping -> torunning ! kcall
    [] looping -> SKIP
    [] TRUE -> looping := TRUE
    END
  END user;

BEGIN
  ready(ch0, ch1);
  running(ch0, ch1, ch2, ch3);
  device(ch3, ch0);
  user(ch2)
END scheduler;
```

Bibliography

[1] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] R. Bagrodia. Synchronisation of Asynchronous Processes in CSP. *ACM Transactions on Programming Languages and Systems*, 11(4):585–597, 1989.

[4] D.C. Barnard. A Solution to the State Explosion Problem During Model Checking. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 1991.

[5] G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics and Implementation. *Science of Computer Programming*, 19:87–152, 1992.

[6] C. Binding. Executing LOTOS Behavior Expressions. Research Report RZ 2118, IBM Research Division, Zurich Research Laboratory, April 1991.

[7] G.V. Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Transactions on Computers*, C-31(3):223–231, March 1982.

[8] P. Brinch Hansen. *Brinch Hansen On Pascal Compilers.* Prentice-Hall, 1985.

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

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

[11] M.C. Browne. An Improved Algorithm for the Automatic Verification of Finite State Systems Using Temporal Logic. In *Proceedings of the Symposium on Logic in Computer Science*, pages 260–266, Washington D.C., June 16-18 1986. IEEE Computer Society Press.

[12] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, (59):115–131, 1988.

[13] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[14] J.R. Burch, E.M. Clarke, K.L McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5-th IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[15] R.W. Butler and G.B. Finelli. The Infeasibility of Experimental Quantification of Life-Critical Software Reliability. In *Proceedings of the ACM SIGSOFT 91 Conference on Software for Critical System*, pages 66–76, New Orleans, Louisiana, December 1991.

[16] M. Chiodo and M. Marelli. Automatic Reduction in CTL Compositional Model Checking. In *Participants proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[17] E.M. Clarke. Personal Communication, June 1993. REX Symposium, Noordwijkerhout, Netherlands.

[18] E.M. Clarke and I.A. Draghicescu. Expressibility Results for Linear-Time and Branching-Time Logics. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 428–437. Lecture Notes in Computer Science, 354, Springer Verlag, 1988.

[19] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of IBM*

Workshop on Logic of Programs, pages 52–71. Lecture Notes in Computer Science, 131, 1981.

[20] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. *Proceedings 10th ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

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

[22] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *Proceedings of the Fifth International Conference for Computer-Aided Verification*. Lecture Notes in Computer Science, 697, July 1993.

[23] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. In *Proceedings of 19th Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 343–354, Albuquerque, New Mexico, January 1992.

[24] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench : A Semantics Based Tool for the Verification of Concurrent Systems. Research report, University of Edinburgh, February 1992.

[25] A. Cohn. Correctness Properties of the Viper Block Model: The Second Level. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 1, pages 1–91. Springer-Verlag, 1989.

[26] P. de Villiers. A Model Checker for Transition Systems. In *6-th Southern African Computer Symposium*, pages 262–275, July 1991.

[27] P. de Villiers and W. Visser. ESML—A Validation Language for Reactive Systems. In *7-th Southern African Computer Symposium*, July 1992.

[28] M. Diaz. Modeling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models. *Computer Networks*, (6):419–441, 1982.

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

[30] G. Donnan and M. E. C. Hull. On Processes, Synchronisation and Redundant Code in Communicating Sequential Processes. *Computer Languages*, 11(3):155–160, 1986.

[31] E.A. Emerson and E.M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[32] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'Not never' Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[33] E.A. Emerson and J. Srinivasan. Branching Time Temporal Logic. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 123–172. Lecture Notes in Computer Science, 354, Springer Verlag, 1988.

[34] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. In *Proceedings of the Third International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 575, July 1991.

[35] H. Everitt. Temporal Logic as an Aid to Validating Communication Protocols. In *Proceedings of the Australian Software Engineering Conference*, pages 293–305, Canberra, May 11-13 1988.

[36] W. Fouché and P. de Villiers. A Reusable Kernel for the Development of Control Software. In *6-th Southern African Computer Symposium*, pages 83–94, July 1991.

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

[38] P.G. Frankl and E.J. Weyuker. Assessing the Fault-Detecting Ability of Testing Methods. In *Proceedings of the ACM SIGSOFT 91 Conference on Software for Critical System*, pages 77–91, New Orleans, Louisiana, December 1991.

[39] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proceedings of the Second International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 531, June 1990.

[40] P. Godefroid and G.J. Holzmann. On the Verification of Temporal Properties. In *Participants Proceedings of the 13-th IFIP Symposium on Protocol Specification, Testing, and Verification*, Liège, Belgium, 25-28 May 1993.

[41] P. Godefroid, G.J. Holzmann, and D. Pirottin. State Space Caching Revisited. In *Participants proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[42] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proceedings of the Third International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 575, July 1991.

[43] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[44] S. Graf, J.-L. Richier, C. Rodrigues, and J. Voiron. What are the Limits of Model Checking Methods for the Verification of Real Life Protocols? In *Proceedings of the International Workshop on Automatic Verification Methods for Finite Systems.* Lecture Notes in Computer Science, 407, June 1989.

[45] K. Hamaguchi, H. Hiraishi, and S. Yajima. Design Verification of a Microprocessor Using Branching Time Regular Temporal Logic. In *Participants proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[46] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[47] D. Harel and A. Pnueli. On the Development of Reactive Systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, New York, 1985.

[48] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

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

[50] G.J. Holzmann. Algorithms for Automatic Protocol Verification. *AT&T Technical Journal*, pages 32–44, January/February 1990.

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

[52] G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. In *12-th International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.

[53] A.J. Hu, D.L. Dill, A.J. Drexler, and C.H. Yang. Higher-Level Specification and Verification with BDDs. In *Participants proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[54] M.E.C. Hull. Implementation of the CSP Notation for Concurrent Systems. *The Computer Journal*, 29(6):500–505, 1986.

[55] Inmos. *Occam 2 Reference Manual*. Prentice-Hall, 1988.

[56] ISO/TC97/WG16-1. *LOTOS: A Formal Description Technique*, 1984.

[57] C. Jard and T. Jéron. On-line Model Checking for Finite Linear Temporal Logic Specifications. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite Systems*. Lecture Notes in Computer Science, 407, June 1989.

[58] C. Jard and T. Jéron. Bounded-memory Algorithms for Verification On-the-fly. In *Proceedings of the Third International Conference for Computer-Aided Verification*. Lecture Notes in Computer Science, 575, July 1991.

[59] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2 edition, 1990.

[60] T.A. Joseph, T. Rauchle, and S. Toueg. State Machines and Assertions: An Integrated Approach to Modeling and Verification of Distributed Systems. *Science of Computer Programming*, 7:1–22, 1986.

[61] G. Karjoth. LOEWE: A LOTOS Engineering Workbench. Research Report RZ 2143, IBM Research Division, Zurich Research Laboratory, June 1991.

[62] G. Karjoth. XFSM: A Formal Model of Communicating State Machines for Implementing Specifications. Research Report RZ 2143, IBM Research Division, Zurich Research Laboratory, June 1991.

[63] G. Karjoth. Implementing LOTOS Specifications by Communicating State Machines. In *Proceedings of the Third International Conference on Concurrency Theory*. Lecture Notes in Computer Science, 630, August 1992.

[64] R.A. Karp. Proving Failure-Free Properties of Concurrent Systems Using Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 6(2):239–253, April 1984.

[65] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

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

[67] Z. Manna and A. Pnueli. The Anchored Version of the Temporal Framework. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 201–284. Lecture Notes in Computer Science, 354, Springer Verlag, 1988.

[68] Z. Manna and A. Pnueli. Completing the Temporal Picture. Research Report STAN-CS-89-1296, Department of Computer Science, Stanford, California 94305, December 1989.

[69] F. Maraninchi. Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by using a Process Algebra. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite Systems*. Lecture Notes in Computer Science, 407, June 1989.

[70] K.L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[71] K.L. McMillan. The SMV System. DRAFT, February 1992.

[72] K.L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in the verification of Asynchronous Circuits. In *Participants proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[73] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[74] J.S. Ostroff. *Temporal Logic for Real-Time Systems*. John Wiley and Sons Inc., New York, 1989.

[75] A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.

[76] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 510–584. Lecture Notes in Computer Science, 224, Springer Verlag, 1986.

[77] D.K. Probst and H.F. Li. Using Partial-Order Semantics to Avoid the State Explosion Problem in Asynchronous Systems. In *Proceedings of the Second International Conference for Computer-Aided Verification*. Lecture Notes in Computer Science, 531, June 1990.

[78] C. Ratel, N. Halbwachs, and P. Raymond. Programming and Verifying Critical Systems by means of the Synchronous Data-Flow Language LUSTRE. In *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems*. ACM Press, December 1991.

[79] N. Rico, G. Bochmann, and O. Cherkaoui. Model Checking for Real-Time Systems Specified in LOTOS. In *Participants proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[80] R.L. Schwartz and P.M. Melliar-Smith. From State Machines to Temporal Logic: Specification Methods for Protocol Standards. *IEEE Transactions on Communications*, COM-30(12):2486–2496, December 1982.

[81] A. Silberschatz. Port Directed Communication. *The Computer Journal*, 24(1):78–82, 1981.

[82] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

[83] D. Talia. Notes on Termination of Occam Processes. *ACM SIGPLAN NOTICES*, 25(9):17–24, 1990.

[84] A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings of the Second International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 531, June 1990.

[85] H. Wong-Toi and D. Dill. Synthesizing Processes and Schedulers from Temporal Specifications. In *Proceedings of the Second International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 531, June 1990.

[86] K.L. Wrench. CSP-i: An Implementation of Communicating Sequential Processes. *Software—Practice and Experience*, 18(6):545–560, June 1987.