

A Model Checker for the LF System

Erick D.B. Gerber



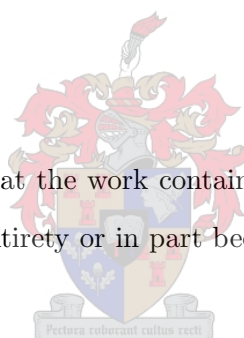
THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH.

Supervised by: Dr. J. Geldenhuys

March 2007

Declaration

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



Signature:

Date:

Abstract

Computer aided verification techniques, such as model checking, can be used to improve the reliability of software. Model checking is an algorithmic approach to illustrate the correctness of temporal logic specifications in the formal description of hardware and software systems. In contrast to traditional testing tools, model checking relies on an exhaustive search of all the possible configurations that these systems may exhibit. Traditionally model checking is applied to abstract or high level designs of software. However, often interpreting or translating these abstract designs to implementations introduce subtle errors. In recent years one trend in model checking has been to apply the model checking algorithm directly to the implementations instead.

This thesis is concerned with building an efficient model checker for a small concurrent language developed at the University of Stellenbosch. This special purpose language, LF, is aimed at development of small embedded systems. The design of the language was carefully considered to promote safe programming practices. Furthermore, the language and its runtime support system was designed to allow directly model checking LF programs. To achieve this, the model checker extends the existing runtime support infrastructure to generate the state space of an executing LF program.

Opsomming

Rekenaar gebaseerde program toetsing, soos modeltoetsing, kan gebruik word om die betroubaarheid van sagteware te verbeter. Model toetsing is 'n algoritmiese benadering om die korrektheid van temporale logika spesifikasies in die beskrywing van harde- of sagteware te bewys. Anders as met tradisionele program toetsing, benodig modeltoetsing 'n volledige ondersoek van al die moontlike toestande waarin so 'n beskrywing homself kan bevind. Model toetsing word meestal op abstrakte modelle van sagteware of die ontwerp toegepas. Indien die ontwerp of model aan al die spesifikasies voldoen word die abstrakte model gewoonlik vertaal na 'n implementasie. Die vertalings proses word gewoonlik met die hand gedoen en laat ruimte om nuwe foute, en selfs foute wat uitgeskakel in die model of ontwerp is te veroorsaak. Deesdae, is 'n gewilde benadering tot modeltoetsing om dié tegnieke direk op die implementasie toe te pas, en sodoende die ekstra moeite van model konstruksie en vertaling uit te skakel.

Hierdie tesis handel oor die ontwerp, implementasie en toetsing van 'n effektiewe modeltoetsers vir 'n klein gelyklopende taal, LF, wat by die Universiteit van Stellenbosch ontwikkel is. Die enkeldoelige taal, LF, is gemik op die veilige ontwikkeling van ingebede sagteware. Die taal is ontwerp om veilige programmerings praktyke aan te moedig. Verder is die taal en die onderliggende bedryfstelsel so ontwerp om 'n model toetsers te akkomodeer. Om die LF programme direk te kan toets, is die model toetsers 'n integrale deel van die bedryfstelsel sodat dit die program kan aandryf om alle moontlike toestande te besoek.

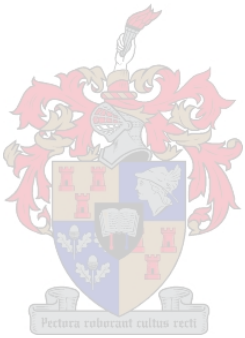
Acknowledgements

I gladly acknowledge the help of several people who made this project feasible:

- My supervisor Dr. J. Geldenhuys, for his excellent guidance and help above and beyond the call of duty.
- Prof. P. J. A. de Villiers for his copious support and guidance, sharing his knowledge and insight on model checking.
- All the members of the Hybrid Lab, past and present, specifically, Leon Grobler, Hanno Bezuidenhout, François Louw, Rudolf Kapp, Jaques Eloff, Riaan Swart, for many a hour discussing LF and related issues, and the many fun hours relaxing together.
- My family and friends, whose support was of unspeakable value during this time.
- The National Research Foundation of South-Africa and Thrip for financial support.

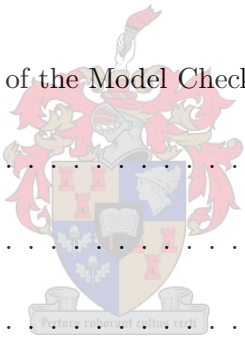
Contents

Abstract	iii
Opsomming	iv
Acknowledgements	v
1 Introduction	1
2 Background	4
2.1 Model Checking	4
2.1.1 Finite Transition Systems	4
2.1.2 Temporal Logic	7
2.1.3 Algorithms for Verification	9
2.1.4 The State Explosion Problem	12
2.2 The LF System	13
2.2.1 The LF Language	15
2.2.2 Actions and Scheduling	19
2.2.3 LF Runtime Support	20



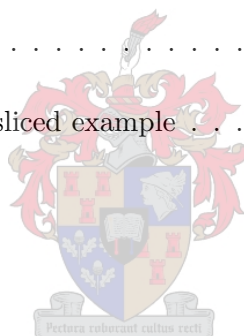
2.2.4	Memory Management and Kernel Structure	23
2.2.5	A Model Checker for LF	23
2.3	Related Work	24
3	Design	27
3.1	Semantic Coverage and Granularity	28
3.2	State Generation and Storage	30
3.2.1	State Representation	31
3.2.2	State Storage	33
3.2.3	Storing Large State Spaces	35
3.3	Temporal Properties	38
3.3.1	Nested Depth-First Searches	39
3.3.2	Strongly Connected Components	41
3.4	Closing Open Systems	43
3.4.1	Manual Methods	43
3.4.2	Automatically Deriving Environments	44
3.5	Model Checking Embedded Applications	46
4	Implementation	48
4.1	System Organisation	49
4.1.1	Module Layout	51
4.1.2	Additional System Calls	52

4.1.3	Defining the System State	53
4.2	Generating States	55
4.2.1	Depth-First Search	56
4.2.2	Selecting Actions	57
4.3	State Storage	61
4.3.1	Local States	61
4.3.2	Global States	62
4.3.3	State Cache	64
5	Evaluation	66
5.1	Establishing the Correctness of the Model Checker	67
5.2	Performance	70
5.2.1	Examples	71
5.2.2	Profile	72
5.2.3	Actions	74
5.2.4	Sorted Channel Queues	75
5.3	Comparison to Other Model Checkers	76
6	Conclusion	80
6.1	Related and Future Work	82
6.2	Remarks	83



List of Tables

1	Some examples used in the Comparison	71
2	Normal LF program vs. Fine grained	75
3	Savings incurred by sorting the queues	76
4	Statistics of several models	78
5	Reachable states of a small sliced example	83



List of Figures

1	State graph of P	5
2	The state graph of $P \times P$	6
3	A Büchi automaton for the property $\Box\Diamond p$	11
4	The LF system.	14
5	A simple producer/consumer example in LF	18
6	The module layout of the LF kernel	24
7	Property process for $\Box\Diamond p$, where p in the example is a Boolean statement	39
8	The Nested Depth-First Search Algorithm	40
9	A general device model	45
10	Memory Organisation of LF Model Checker	50
11	The module layout of the model checking kernel	52
12	The Depth-First Stack	55
13	The Depth-First Search Engine	58
14	The BNF of the CHOICE operator	60
15	The structure of the state vector	63

16 A snippet of the SCC detection algorithm 69

17 Profile of the system 73



Chapter 1

Introduction

Reactive systems is a class of software that is characterised by a continuous interaction with the environment, and unlike transformational programs, they do not compute a final answer. Embedded systems are examples of reactive software that are constructed for specialised hardware that form part of larger electromechanical systems, or consumer electronic goods. The correctness of these systems is important because they are often mass produced, and correcting software defects may be costly. Moreover, embedded software is frequently used to control expensive hardware, and software defects may cause damage to these systems and, in some extreme cases, injury or loss of life.

Formal methods is a rigorous application of mathematics to the development of software. A great advance in formal methods is the advent of computer-aided verification techniques, one of which is model checking. A model checker is a program that finds, without user assistance, violations of desirable properties in the description of software systems. The relative success of model checking can be attributable to several factors: once the user has specified the system and its properties, the verification is automated; and if there is a violation of the property the model checker can produce a witness to the erroneous behaviour.

A model checker requires at least three ingredients: (1) a formalism to express the software system, (2) a formalism to describe properties of the system, and (3) an algorithm to illustrate that the software satisfies the property. In many model checkers an abstract modelling

language is used to describe software systems, and some form of temporal logic is used for the specification of properties. The choice of an model checking algorithm, however, depends on a number of factors.

Two popular approaches to model checking are symbolic and explicit state model checking. Explicit state model checking relies on enumerating all possible configurations (or states), of the software system, one at a time, while applying the model checking algorithm to each of the states. Symbolic model checking does not generate individual states, but calculates a fixed point of states, and applies the model checking algorithm to all the states in the fixed point at the same time.

In the past decade the model checking community has invested effort in model checking programs directly. This is a step to make model checking more ubiquitous in the development life cycle of software. A side-effect of model checking programs is that it drives research to better handle large state spaces due to the inherent complexity of programming languages.

The LF System

The LF system is designed to implement small embedded systems [74, 34, 70]. The system comprises a concurrent programming language together with a runtime environment. The language was designed to promote safe programming practices by eliminating features such as dynamic memory allocation and pointers, and includes features such as strong typing and array index bounds checking. One of the main design goals of the LF project was that the system must support model checking natively. That is, a model checker will form an integral part of the execution environment.

Rather than translating an LF program into an intermediate form suitable for model checking, the executable code generated by the compiler serves as input to the LF model checker. A model checking kernel drives and observes the hardware to execute the program. At key points in the execution the model checker takes control, and in this way it enumerates all possible program states. An automata-theoretic model checking algorithm is applied on-the-fly.

The goal of this thesis is to derive and evaluate an effective model checker which can directly

model check the executing LF program. This involves three main tasks:

1. integrating the model checker into the existing runtime kernel;
2. generating program states, and detecting revisited states; and
3. dealing with open systems, and model checking device drivers.

Thesis outline

The rest of this thesis is structured as follows:

Chapter 2: Background provides a brief introduction to the main concepts of model checking and an overview of the LF system (in particular the language and runtime system) and the requirements of the model checker. The chapter concludes with a brief discussion of other approaches to deriving verified programs and other program model checkers.

Chapter 3: Design derives a design for the model checker. First we establish what has to be verified and then discuss the data structures and algorithms that can be used to implement such a model checker. The key design decisions and their effect on the efficiency of the model checker is discussed. The chapter ends with a discussion of model checking in resource-poor environments.

Chapter 4: Implementation outline the implementation of the LF model checker and focuses on two main issues in the LF model checker: generating new states and the implementation of state storage.

Chapter 5: Evaluation presents the results of several experiments that firstly attempt to illustrate the correctness of the state generator and cycle detection algorithm, and secondly establish how the design decisions affect the performance of the model checker.

Chapter 2

Background

This chapter discusses model checking in brief, examines the LF language and runtime system, and discusses how to apply model checking to this system. It concludes with a brief discussion of similar work and other approaches which use model checking to create verified programs.

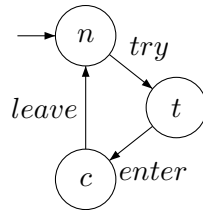
2.1 Model Checking

As mentioned in the Introduction, three key ingredients are required for model checking: (1) a formalism to describe the program, (2) a formalism to describe the property, and (3) an algorithm to perform the verification. In this section each of these aspects will be examined in turn.

2.1.1 Finite Transition Systems

A necessary requirement to reason about a concurrent program is a formal description of the system and its behaviour. Several formalisms have been developed to describe the behaviour of programs but we focus here on just one common approach, namely Finite Transition Systems (FTS).

An FTS represents a system as a finite set of states and transitions. A transition relation describes how successor states are generated. Formally, a FTS is a tuple $M = (S, T, R, q)$

Figure 1: State graph of P

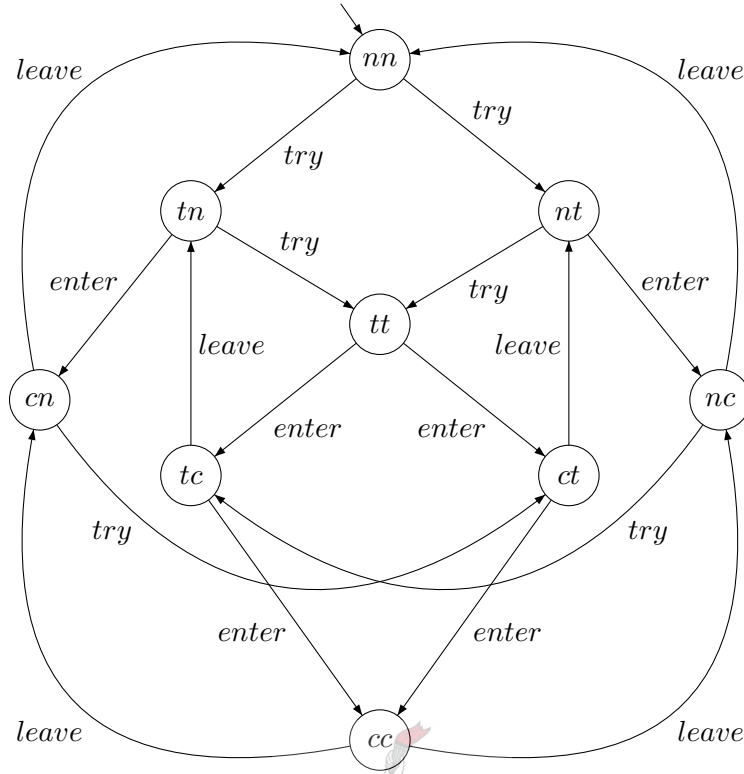
where:

- S is the finite set of states. Each state is a canonical description of the system at a specific moment in time.
- T is the finite set of transitions. A transition is an atomic step that transforms the system from one state to another.
- $R \subseteq S \times T \times S$ is a transition relation that maps each state and transition to a successor state.
- q is the initial state.

If $(s, t, s') \in R$, then the transition t is said to be enabled in state s , executing transition t in s yields state s' . We abbreviate this as $s \xrightarrow{t} s'$. The entire set of enabled transitions in state s is $en(s)$. An execution trace is any infinite sequence of states $\delta = s_0s_1s_2\dots$ such that for all $i \geq 0$, there is a transition t_i such that $s_i \xrightarrow{t_i} s_{i+1}$. A state s' is reachable from state s if there exists a finite sequence of states $\delta = s_0s_1\dots s_n$ so that $s = s_0$ and $s' = s_n$. In short, $s \xrightarrow{\delta} s'$.

Figure 1 shows an FTS for process P , which continuously tries to enter a critical section. The process has three distinct states $\{n, t, c\}$ for non-critical, trying and critical, and three transitions $\{try, enter, leave\}$. The transition relation is $R = \{(n, try, t), (t, enter, c), (c, leave, n)\}$ and $q = n$.

The previous example contained only one process. Usually concurrent systems comprise many such processes. An FTS which describes the behaviour of the complete system is the product

Figure 2: The state graph of $P \times P$

of all its components. This is an FTS with the set of states $S \subseteq S_1 \times S_2 \times S_3 \dots S_n$, transitions $T \subseteq \bigcup_0^n T_i$ and a transition relation $R \subseteq S \times T \times S$. Exactly which transitions are allowed in the product is dictated by the semantics of the system. For example, in many process algebras, transitions with the same alphabet symbol can in some sense cooperate and execute simultaneously. This is referred to as the synchronous product [38(Chapter 2)]. However in this thesis we restrict ourselves to the asynchronous product, which we define as follows: $(s_1, s_2, s_3 \dots s_n) \xrightarrow{t} (s'_1, s'_2, s'_3 \dots s'_n) \in R$ if and only if for some $0 < i \leq n$, $(s_i, t, s_i) \in R_i$ and $s_j = s'_j$ for $i \neq j$.

The asynchronous product of two processes of type P is the FTS shown in Figure 2

The labels of a vertex is the current state of all the processes at that instance, for example, state ct is a state where one process is currently executing inside its critical section and the other is trying to gain access to its critical section.

The FTS of Figure 2 can be easily transformed to a Kripke structure by applying a labelling function L to the states. Kripke structures can be seen as an extension of FTS's where each state is associated with a set of true propositions. For instance, if we are interested in ensuring that the two copies of P are never in their critical states at the same time, we define the labelling function $L(cc) = \emptyset$ and $L(s) = \{OK\}$ for $s = \{nn, cn, nc, ct, tc, nt, tn, tt\}$ where OK is an atomic proposition that is true if and only if the mutual exclusion property holds.

2.1.2 Temporal Logic

Having discussed a formal representation of concurrent systems, we need a formalism to reason about the correctness of a software system. The basis laid by Floyd and Hoare for the verification of sequential programs, using preconditions and postconditions [37], was later extended to accommodate parallel programs by Owicki and Gries [58]. These correctness proofs are rather complicated, are performed manually (for the most part), and only works well when the concurrent system describes an input-output relation. In other words, they are not well suited to reactive systems.

Temporal logics introduced by Pnueli [64] to Computer Science allows us to reason about the relative ordering of events over time without explicitly modelling time. Temporal logics extend propositional logic by adding operators that refer to the sequence of states.

There are two varieties of temporal logic: linear time and branching time. Linear time temporal logic, such as (Propositional) Linear Time Logic (or LTL for short) [53], is concerned with single executions. LTL can express formulas that must hold for all paths starting at the initial state. Branching time temporal logics such as Computational Tree Logic (CTL) [18], can express properties that should hold in all possible paths, and properties that should hold in at least one possible future path.

The discussion of the relative merits of branching and linear time temporal logic, specifically LTL and CTL, is nearly as old as these paradigms themselves [49]. It was shown in [19] that the two logics are incomparable as there are formulas that can be expressed in LTL which are not in CTL and vice versa, and introduced a new encompassing temporal logic CTL* which

includes both LTL and CTL. In [48] the authors show that although in the worst case CTL model checking is computationally cheaper, in general LTL and CTL model checking performs on par. LTL is often preferred to express properties of software systems due to its notational simplicity. In the next section we give a brief introduction to LTL and its semantics.

LTL

For the purpose of this thesis, an informal description of LTL suffices; for a more formal discussion of LTL the reader is referred to [54] and [20]. LTL inherits all of propositional logic; atomic propositions (p, q, r, \dots), binary operators ($\vee, \wedge, \Rightarrow, \Leftrightarrow, \dots$) and negation (\neg), and extend it with modal operators: the unary temporal operator \bigcirc and the binary temporal operator \mathcal{U} . More precisely, if P is a set of atomic propositions then LTL formulas are defined inductively as follows:

- every member of P is a formula, and
- if ϕ and ψ are formulas, then so are $\neg\phi, \phi \wedge \psi, \bigcirc\phi, \phi \mathcal{U} \psi$.

An interpretation of an LTL formula is an infinite word $\xi = a_0, a_1, a_2, \dots$ over the alphabet 2^P , the power set of P . The suffix of ξ starting at a_i is written as ξ_i and $\xi \models \phi$ denotes the fact that ξ satisfy formula ϕ . The semantics of LTL can then be defined as follows:

- $\xi \models q$ if $q \in a_0$ and, for $q \in P$
- $\xi \models \neg\phi$ if not $\xi \models \phi$
- $\xi \models \phi \wedge \psi$ if $\xi \models \phi$ and $\xi \models \psi$
- $\xi \models \bigcirc\phi$ if $\xi_1 \models \phi$
- $\xi \models \phi \mathcal{U} \psi$ if there is an $i \geq 0$ such that $\xi_i \models \psi$ and $\xi_j \models \phi$ for all $0 \leq j < i$

Additional operators are defined in terms of these basic operators:

- $\diamond p \equiv \top \mathcal{U} p$ — (sometimes) the proposition p holds in the future

- $\Box p \equiv \neg \Diamond \neg p$ — (always) the proposition p holds globally
- $p\mathcal{R}q \equiv \neg(\neg\phi \mathcal{U} \neg\psi)$ — (release) p holds either globally or until q holds

LTL properties have been classified into a hierarchy expressing similar structural or behavioural properties. Most notably, Lamport [49] proposed that temporal properties describe either *safety* or *liveness* properties. Safety properties guarantee that something undesirable never happens, for instance that a buffer never overflows. Liveness properties express desirable behaviours that must eventually occur.

2.1.3 Algorithms for Verification

Early model checkers employ a quite naive approach. First the entire state graph is generated, and then induction over the temporal logic formula is used to show that the system satisfies the temporal property [18]. The available computer hardware resources at the time, severely limited the size of the systems that could be verified. Nevertheless, the method is sound.

Over the past 25 years, available hardware has improved and more sophisticated algorithms have been developed. Nowadays, two popular approaches to model checking are symbolic model checking and automata theoretic model checking.

Symbolic Model Checking

Symbolic model checking [5] calculates the fixed point of the FTS transition relation to compute the reachable states. This is combined with the fixed point characterisation of temporal logic formulas to calculate all states that satisfy the specification. In theory, the fixed point algorithm does not specify a specific data structure, and can be implemented in a variety of ways. In practise however, symbolic model checking is generally associated with binary decision diagrams (BDD) [3]. This data structure stores boolean functions in a dense way. BDD's are particularly well suited to synchronous systems, such as digital circuits. Therefore symbolic model checking is usually applied in the verification of hardware systems. Unfortunately, when dealing with software verification this approach fares less well,

and therefore we will not deal with it further. For interested readers [5], [7] and [55] provide a good starting point.

Automata-Theoretic Model Checking

Finite state automata play a central role in many fields of Computer Science, and it is no surprise that they are useful in model checking. While “standard” automata describe languages with a finite or infinite number of words, each word is finite in length. To use automata in model checking, we need to turn to ω -automata, and in particular, Büchi automata [27], that accept words of infinite length.

A Büchi automaton is an ordered tuple $\mathcal{B} = (S, q, \Sigma, \Delta, F)$ where:

- S is a finite set of states.
- $q \in S$ is the initial state.
- Σ is a finite alphabet.
- $\Delta \subseteq S \times \Sigma \times S$ is a transition relation.
- $F \subseteq S$ is the set of accepting states.



An infinite word $a_0, a_1, a_2, \dots \in \Sigma^\omega$ is accepted by a Büchi automaton if and only if (1) there is an infinite sequence of states $\rho = s_0, s_1, s_2, \dots \in S^\omega$ such that $s_0 = q$ and $(s_i, a_i, s_{i+1}) \in \Delta$ for all $i \geq 0$, and (2) at least one state $s \in F$ occurs infinitely often in ρ . LTL formulas can be transformed to equivalent Büchi automata [75, 76, 27]. In this case, $\Sigma = 2^P$, where P is a set of atomic propositions. An example of a Büchi automaton for the property $\Box \Diamond p$ is shown in Figure 3.

One algorithm for constructing Büchi automata from LTL formulas is the Gerth, Pered, Vardi and Wolper algorithm [27]. The algorithm requires structuring the formula into a normal form, and the Büchi automaton is constructed by decomposing the formula using a fixed point definition of the LTL operators.

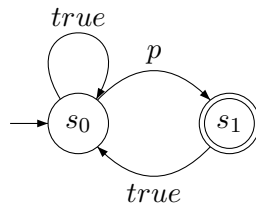


Figure 3: A Büchi automaton for the property $\Box\Diamond p$

The Büchi automaton is central to the automata-theoretic approach. In essence, the algorithm relies on viewing both the behaviour of the system and the correctness specification as Büchi automata, and computing their intersection. The intersection of the two automata represent those execution traces that conform to the specification. Therefore composing the system with an LTL formula does not illustrate those execution traces that could violate the property.

To remedy this, the LTL formula is first negated before being transformed into a Büchi automaton and combined with the system. The resulting intersection Büchi automaton then represents all the execution traces that adhere to the negated formula, and violates the original formula. In short, if the language of the intersection is not empty, the property is violated. An algorithm for automata theoretic model checking for LTL was first described in [76].

Determining the emptiness of the intersection automaton is the main goal in automata-theoretic model checking. As we have seen above, any accepting sequence of states has to contain at least one state from F infinitely often and, since S is finite, at least one cycle that contain states from F is necessary. In other words, the model checking problem has been reduced to the graph-theoretic problem of finding cycles that contain accepting states.

Explicit State On-the-Fly Model Checking

In on-the-fly model checking the model checking algorithm is applied to states during the construction of the state graph instead of first building the state graph of the model and then computing the intersection of the model and the property. One benefit of the on-the-fly method is that if the model contains a violation of a property, only the states that illustrate the violation need to be explored. The on-the-fly construction of the state graph uses a systematic search — for instance a depth-first search — of the state graph.

Exploring a state several times does not alter the result of the verification. In order to avoid such redundant work states which have been explored is explicitly stored in some data structure. If the search reaches a state for the second time, it can be ignored as the model checking algorithm has already been applied to it.

Automata-theoretic model checking is usually applied in an on-the-fly manner, constructing the intersection of the Büchi automaton and state graph during the construction of the state graph. A transition of the Büchi automaton is executed for each transition of the system. If there is no viable transition from the Büchi automaton in the current state, this path cannot violate the property and it need not be investigated further.

Two basic techniques for on-the-fly model checking has been proposed. One class of algorithms relies on finding strongly connected components (SCCs) in the state graph. If there exists an SCC reachable from the initial state which contains one or more accepting states, the property is violated.

The second class of algorithms is based on the Coucoubertis, Vardi, Wolper, Yannakakis [12] algorithm, which use one depth-first search to find acceptance states and a second (possibly nested) search to find a cycle containing these accepting states.

2.1.4 The State Explosion Problem

One of the greatest obstacles in model checking is the “*state explosion*” problem: The number of states contained in any interesting system is typically huge and increases exponentially as the number of components in the system increases linearly. The source of the state explosion is the combinatorial number of interleavings and interactions between the various concurrent components.

For example, if we have a system of n independent processes each with k local states, the number of reachable states is k^n . If the number of processes is doubled to $2n$, the number of states is squared to k^{2n} . If the processes interact in some way the number of reachable states may be smaller, but the trend almost always remains exponential.

This exponential explosion in states presents a serious challenge to model checking. Not only

does it take longer to investigate more states, but if previously visited states are recorded it requires a significant amount of storage space to complete the verification. A good survey of the state explosion problem and methods used to combat problem is [73].

Several techniques have been devised to deal with the state explosion problem:

- Partial order methods [28, 63, 62, 72]: The basis of partial order techniques is that several interleavings may be equivalent with respect to the correctness of a property. A single state in a model may be reached by several paths, which differ only slightly in the ordering of the transitions. By ignoring some redundant interleavings and only exploring representative interleavings, fewer transitions and states need to be explored. Typical examples of these techniques are persistent sets [72] and sleep sets [28].
- Abstraction [8, 14, 50]: The gist of abstraction techniques is to eliminate the irrelevant detail in a model. One well-understood abstraction technique is predicate abstraction. Predicate abstraction maps the variable space onto a much smaller set while retaining the essential behaviour (or over approximating it slightly) of the system. For example, an integer variable which assumes a large range of values can be mapped to the set $\{positive, zero, negative\}$, which will dramatically reduce the number of reachable states.
- Symmetry reduction [21, 69]: Often software systems exhibit a large degree of symmetry. If a system consists of several identical components, each of the orderings of the components yields different states. By fixing a single canonical ordering of components, these states can be identified as equivalent.

2.2 The LF System

The LF system was designed to implement small- to medium-sized embedded applications. Embedded systems are often required to perform safety-critical tasks. They often need to operate in a real-time environment, but even when this is not called for, they need to be

highly efficient. The result is that even small systems can have a complex concurrent design which makes it difficult to reason about their correctness.

The design of the LF system is centred around this problem and the use of model checking as a possible solution. To this end the design incorporated several key features from the start.

1. To avoid discrepancies between the actual program and its formal description, model checking is applied directly to the executable code generated by the compiler. An LF program can therefore either be executed on the LF kernel or it can be explored on the model checker.

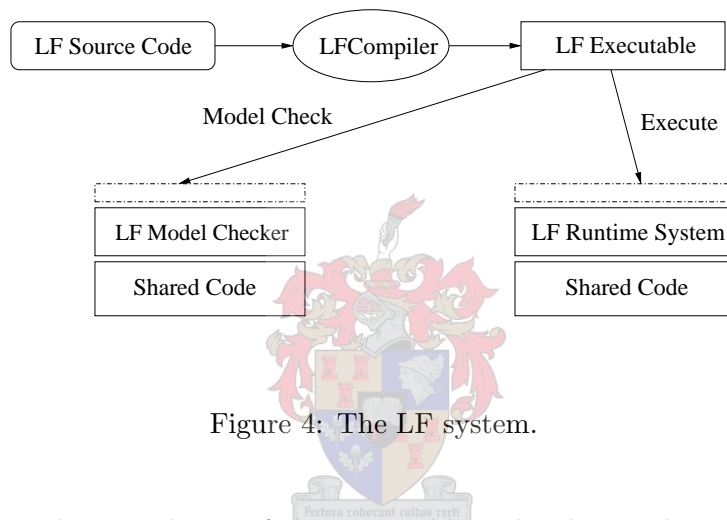


Figure 4: The LF system.

2. By increasing the granularity of the executable code, the number of interleavings are reduced. Rather than executing individual instructions, LF programs consists of blocks of instructions, that are executed as a unit. Section 2.2.2 describes this idea in detail.
3. The language is designed to promote safe programming practices and to avoid features that would make model checking difficult or expensive. In particular, the language
 - is based on the clear intuitive syntax of the Oberon programming language [57, 52] (easily read and understood code is easier to debug);
 - is strongly typed to eliminate simple programming errors;
 - eliminates dynamic allocation of memory and pointers, which in turn eliminates many intricate programming errors and simplifies memory management;

- relies on synchronous message passing for interprocess communication (conceptually simpler); and
- limits the interaction between processes to only the interprocess communication to simplify reasoning about the behaviour of the system.

2.2.1 The LF Language

The LF system has gone through several iterations before arriving at the language that we will discuss. Originally the system was conceived to be an embeddable version of the Joyce programming language by Brinch Hansen [59]. It supported typed pointers, placing variables at absolute addresses and various bit manipulation and input-output operations, but lacked modularisation and reference parameters for processes.

The use and misuse of pointers hampered early attempts to model check LF [4]. The language was therefore redesigned to eliminate the above-mentioned problems. Removing dynamic memory allocation and pointers simplifies the model checking of LF considerably.

Language Constructs

The unit of compilation in LF is a *module* and every program comprises one or more modules. Modules contain declarations of processes, channels and global variables. Each module has an interface that consists of the process and channel definitions it exports, and a module can indicate its use of another module with the *IMPORT* statement.

Processes in LF may have both pass-by-value and pass-by-reference parameters, local variables and local channels. All scoping rules in LF are the same as in the Brinch Hansen languages. Process definitions may be nested, and recursion is allowed. Processes are instantiated either to run concurrently with the parent as a concurrent processes, or to block the parent until the child completes. The latter are known as run-to-completion processes. Run-to-completion processes allow LF to provide a feature similar to procedures in other languages. One important restriction is that concurrent processes in LF are not allowed to share common variables.

LF provides signed and unsigned integers, bit sets, a single-byte character and a Boolean type. In addition to these base types, LF supports two structured types, arrays and records. Strict typing rules are applied during assignments, parameter passing and interprocess communication, which eliminates some programming errors. Explicit type casting operators are supplied to temporarily break the type rules during expression evaluation. Control structures such as *IF*, *WHILE* and *REPEAT* behave exactly as similar structures in other languages such as Oberon.

Interprocess communication is provided by means of synchronous, bidirectional, unbuffered message passing. LF provides a channel construct, which may declare one or more message types that can be transmitted over the channel. A message type can be either a pure signal with no associated data, or a complex message type that has both a signal and data. Channels are allocated during runtime, and references to channels are stored in special objects called ports. Ports may be declared locally and passed via parameters to child processes, or declared globally, allowing all active processes to share the channel.

Message passing in LF is anonymous: the sender or receiver does not directly name its counterpart. A process that wishes to communicate over a channel, may communicate with any of the processes connected to that channel waiting to send or receive the same message type. A sender or receiver blocks until a suitable partner is identified and the message is transferred.

LF provides a powerful extension to the message passing system with the *SELECT* statement. This control structure is somewhat similar to the *CASE* and *switch* statements of other languages. A *SELECT* has one or more *WHEN* clauses, each consisting of a single send or receive command, a Boolean guard, and a statement sequence. A *WHEN* clause is enabled if and only if the communication command is viable (a willing partner is available) and the Boolean guard evaluates to true after the communication command executes. The semantics of a *SELECT* is that it blocks until one or more of its *WHEN* clauses are enabled. It then makes a non-deterministic choice among the enabled clauses and executes that clause's communication statement and statement sequence.

One of the guiding principals of the LF language is that no two concurrently executing processes may share variables. However, as the language evolved, this constraint was relaxed in order to make the language more expressive. Shared variables are also more effective to transfer large volumes of data between processes than synchronous message passing. Relaxing the language semantics in this case was justified by the efficiency gains, and the fact that model checking could be used to find race conditions.

Example

An LF version of the classic producer/consumer program is shown in Figure 5. The program starts with the `MODULE` keyword, followed by its name `ProdCons`. In line 2, it imports the module `Out`, which it will use to display values. It declares a channel `Msg` with two message types: a pure signal called `end`, and a signal `data` that carries an associated integer value. Line 4 declares `q` as a port object that refers to a channel of type `Msg`. The type and variable declarations are followed by two process definitions, `Producer` and `Consumer`. Both processes accept a port variable of type `Msg` as parameter. The process definitions are followed by the module initialisation code from line 27 onwards.

Process `Consumer` follows the same basic layout as the process `Producer`: a single loop with an embedded communication statement. Instead of a simple receive statement a `SELECT` is used. The `SELECT` statement in line 21, allows receiving several message types over one or more channels. In this example, the select contains two `WHEN` clauses, one that accepts and receives the signal `data` and a value in variable `x`. The second `WHEN` clause accepts the signal `end` which indicates the end of a stream of values from the producer. The `Out.Int` statement in line 22 invokes the process `Int` from the module `Out` as a run-to-completion process.

The `NEW` keyword in line 29, creates a new channel and stores a reference to the channel in `q`. In lines 30 and 21 two processes are instantiated, the keyword `CREATE` indicates that the processes should run concurrently with the module initialisation code. Absence of the `CREATE` keyword, indicates that the process should be instantiated as an run-to-completion process. Both processes are passed the `q` variable as parameter, which allows them to communicate

```
1  MODULE ProdCons;
2  IMPORT Out;
3  TYPE Msg = [data(INTEGER),end];
4  PORT q : Msg;
5
6  PROCESS Producer(out : Msg);
7  VAR  i : INTEGER;
8  BEGIN
9      i := 0;
10     WHILE i < 100 DO
11         out!data(i); INC(i)
12     END;
13     out!end
14 END Producer;
15
16 PROCESS Consumer(in : Msg);
17 VAR  x : INTEGER;  more : BOOLEAN;
18 BEGIN
19     more := TRUE;
20     WHILE more DO
21         SELECT
22             WHEN in?data(x) THEN Out.Int(x)
23             WHEN in?end THEN more := FALSE
24         END
25     END
26 END Consumer;
27
28 BEGIN
29     NEW(q);
30     CREATE Producer(q);
31     CREATE Consumer(q)
32 END ProdCons.
```



Figure 5: A simple producer/consumer example in LF

over the newly created channel.

The execution of the program sees the module initialisation code of all the modules being imported and executed. In this example the initialisation code of the imported module `Out` is executed transparently so there is no need for the user to explicitly initialise it.

2.2.2 Actions and Scheduling

LF attempts to limit the state explosion in a novel way: the compiler translates several adjacent statements into one block called an *action*. The runtime system ensures that actions are executed atomically. Interrupts that arise during the execution of an action are recorded but only serviced after the action has finished executing. Structuring the LF system to rely on non-preemptive scheduling rather than preemptive scheduling has several benefits, both when viewed from a model checking perspective as well as from an embedded application perspective.

The context of a process is a data structure that an operating system uses to store the current state of a process. The context contains scheduling information, memory allocation details, and other information required to restart a process when interrupted. In a preemptive scheduling strategy, an executing process may be interrupted between any two machine instructions to make way for another process. In this case, the context of a process must be large enough to record all the registers in use before the process was interrupted. This has two implications: for the operating system a larger context entails slower context switches; for model checking the context of a process forms part of the state and larger states means that fewer states can be stored. Model checking preemptive systems require potentially exploring all interleavings of machine instructions. In all but a few small cases, this is impractical.

In a non-preemptive scheduling strategy, predetermined rescheduling points dictate when a process can be interrupted. The compiler can structure code in such a fashion that, at each scheduling point, no register is in use. This limits the context of the executing process. This approach may seem restrictive in general, but it holds several advantages for embedded systems: greater control over when a process can be interrupted means that a developer has greater control over the real-time behaviour of the system. It also leads to faster context

switching. Other benefits include that many potential race conditions are eliminated; and the use of critical sections, in all but a few cases, become unnecessary.

LF supports non-preemptive scheduling, executing one action for a process when scheduled. The scheduling is provided by a loadable kernel module which interacts with the rest of the kernel through a well-defined interface. Currently, the LF kernel provides two basic schedulers: a Round-Robin(RR) and Earliest Deadline First (EDF) scheduler.

Length of Actions

Originally the compiler was conservative when determining the lengths of the actions, and limited the number of statements grouped together in an ad-hoc manner. To avoid unbounded loops never yielding the processor, the statements inside a loop were grouped together as one action and each iteration of a loop included at least one call to the scheduler. Unfortunately, the length of the actions was too conservative and often only a handful of statements were grouped together which resulted in a much higher than expected scheduler overhead [70].

During the development of the runtime kernel and model checker, the compiler was modified to relax the constraints on the length of the actions. Statement sequences are broken into actions only at communication statements. This small change considerably improved the efficiency of the LF kernel, but not without a penalty. A major problem is that entire loops can be contained in a single action. For small loops this represents a significant improvement. However, large loops without any communication statements may hold the processor for too long. To mitigate this problem, the RESCHEDULE statement was added to the language. Its effect is to voluntarily yield control to the scheduler, thus allowing the programmer to control the length of actions.

2.2.3 LF Runtime Support

The target of the LF system is the development of software for resource-poor embedded environments. It is therefore critical that the kernel that supports the language must be small and efficient. The kernel provides support for process management and interprocess

communication by message passing as well as interrupt management. As an extra design requirement, the kernel provides a harness for the model checker.

Support for Processes

The kernel provides support for the creation, termination, and scheduling of processes. Processes are instantiated as either run-to-completion or concurrent processes, and each process in the LF system is allocated an *activation record*. An activation record is a single contiguous block of memory that records the state of a process. Specifically record contains:

- the size of the activation record (the combined size in bytes, of the fixed activation record header, and the space for local variables);
- the unique process identifier;
- the pool identifier (which determines the type of process and is used for efficient memory management);
- a reference to the parent, first sibling and first child processes;
- a reference to the first channel associated with the process;
- the next action to be executed by this process;
- the current process status, either Blocked, Ready, Sending, Receiving, Selective receiving;
- a reference to the data to be copied during the communication;
- scratch space for registers during actions;
- a static link, the process's nesting parent; and
- space for local variables.

An activation record contains exactly the state of a process. All the activation records of all the processes describe the state of the LF system at any point in time.

Interprocess Communication

A structure similar to the activation records for processes is allocated for each of the channels created in an LF program. This is called a *channel record*. The channel records are used by the kernel for efficient implementation interprocess communication. Strictly speaking implementing synchronous message passing does not require such data structures since messages can be directly transferred from one process to another. The channel record is an optimisation, storing a list of blocked processes that are waiting to communicate on the channel. Instead of scanning all the blocked processes to find a suitable partner, it is only necessary to remove an blocked process from the list.

Each message type that the channel defines has three separate separate queues, one for senders, one for receivers, and the third selective communication. Only the head and tail of the queue is stored in the channel record. The queues are formed by a linked list threaded through the activation records using a specialised field.

The main efficiency bottleneck of the kernel was identified as the support for the synchronous message passing primitives in the language. For this reason, several special cases of message passing were identified and separately optimised in the kernel. The interprocess communication system consists of several system calls optimising each of the five different scenarios: send and receive a signal and at most one double word of data, send and receive with more than a double word of data, polling send and receive with at most one double word of data, polling send and receive with more than a double word of data, and send with an expression that first has to be evaluated. The compiler differentiate between the different cases and generates the appropriate system calls in the program. To improve the efficiency of the **SELECT** construct the kernel implements selective communication statements in such a manner a communication statement in one **SELECT** may not synchronise with a communication statement in another **SELECT**.

2.2.4 Memory Management and Kernel Structure

The LF kernel provides only the most necessary support for process management and inter-process communication. In order to keep the memory management simple, the memory of the target machine is viewed as a linear address space. All the entities in the LF system share this space. General purpose operating systems require stringent memory protection techniques to ensure that one process does not corrupt the address space of another. However, the LF compiler ensures the protection, and little effort is required on the part of the kernel.

The kernel provides minimal memory management, only allocating and deallocating activation- and channel records. Allocation of memory for LF processes is based on a technique described in [35], as the *Quick Fit* allocation scheme. The available memory is viewed as a central pool of activation records. When a process is instantiated, an activation record is selected from the pool, and when it terminates, returned to the pool. For complete details, see [34].

The LF runtime system consists of a boot loader, the kernel, and one or more LF modules that constitute the program. The kernel of the LF system consists of 14 modules which can be arranged into three categories: System Calls, Scheduling, and Kernel Drivers. The kernel drivers provide the necessary support to initialise the hardware and execute the runtime system, as well as minimal input/output. For a rough layout of the modules in the LF kernel see Figure 6. The runtime system provides 26 system calls, which can be categorised into: process management (8), communication (14), and input/output (4).

2.2.5 A Model Checker for LF

Recent trends in model checking [36, 39], attempt at making model checking more ubiquitous in the development life cycle of software. Instead of building abstract models of software for verification a model checker is applied directly to the implementation.

Our approach to software model checking is to directly model check LF programs. Instead of translating an LF program into an intermediate form suitable for model checking, the compiled LF program serves as input to the model checker. The executable that the compiler

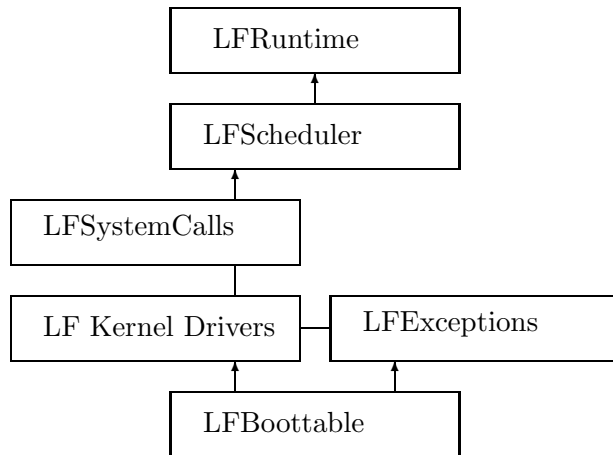


Figure 6: The module layout of the LF kernel

generates can either be run (as usual) on the LF kernel, or model checked by the LF model checker.

In order to avoid discrepancies between the execution of a program on the normal LF kernel and the model checking kernel, most of the existing kernel framework is shared. The model checking kernel, however, must be able to explore all the possible interleavings of the actions of LF processes. This requires abstracting some of the kernel. Specifically the scheduler is replaced by a state generator.

One pitfall in integrating a model checker into the kernel is the number of system calls that the kernel provides which each solve the same problem for different scenarios, and the fact that most of the system calls are implemented in the Intel 386 assembler language.

2.3 Related Work

A correct design is a crucial element in the development of any software. Several formal description techniques have been developed over the years to aid software engineers in formulating designs, and reasoning about them in a systematic way. Model checking has historically been viewed as a design tool. A specification is interpreted and a model representing the design is constructed and checked for several properties. If any errors are encountered in the

model, the model is revised to remove the errors, and checked again, until the design is free of errors. The next phase in the process is to translate this model into an executable implementation. This translation is, for the most part, a manual process and prone to introducing new errors, even errors already eliminated from the model.

Several approaches have been proposed to overcome this specification gap. These include automated translation from the model, extracting models from the implementation, and in recent years, model checking the implementation code directly. We will examine each of these approaches in turn.

Automated Translation

Manual translation of models into implementation is at the mercy of the fallible human translator. Simple errors introduced during translation can invalidate the entire verification of the design. An example of such an error is swapping the order in which locks are acquired to access shared resources.

The first approach to easing the translation from model to implementation is to structure the modelling language as close to an implementation language as possible. The *Promela* specification language [46] was designed with this as one of its guiding principals. Löffner [65] implemented an automated translator for Promela into equivalent C++ code. The approach taken in [4], was to translate Promela into LF with the aid of tabled actions representing equivalent Promela actions.

Translation from models into code still has several drawbacks. Most notably, a developer needs to be highly skilled and knowledgeable in the modelling language and system to be able to effectively use these tools.

Extracting Models from Code

Extracting abstract models of programs is a very popular technique used to verify programs. The approach has several benefits: developers are often reluctant to spend time on model checking, often prior implementations need to be verified, and several automated abstraction

tools are available. Examples of such tools are: Feaver [47], that extracts Promela models from ANSI C code; and Bandera [10], that extract Promela models from Sun Microsystems Java code.

When extracting a model from an implementation, it is important to ensure that all the behaviours, relevant to the property being verified, are preserved. Model extraction is an exercise in abstraction, since a lot of irrelevant detail needs to be removed from the implementation. Abstractions of programs often contain many more behaviours than the original, because the abstraction process adds some nondeterminism when removing detail. Model checking an abstract representation of a programs often yield false negatives. This in turn can be countered by a refinement of the abstraction until a more realistic model is achieved.

Model Checking Code

The increase in the processing power and memory of modern computers and advances in model checking have paved the way for research into model checking implementation code directly [39].

The second generation Java PathFinder (JPF2) [77, 78] is a well-known tool that can verify Java implementations. The JPF2 system uses a tailor-made Java virtual machine that supports all the Java bytecodes, and therefore all pure Java code. JPF2 incorporates a number of tools that enable it to handle the complexity contained in real programs. To relieve the subsequent state explosion, partial orders, symmetry reduction, static analysis, abstraction and heuristic based searches are used.

Another approach to the problem of checking code is that of VeriSoft [32]. Reasoning that the state of an executing program is too complex to efficiently manipulate, Verisoft does not store any states at all. Partial order techniques and a depth-limited search is used to generate a “state space” for C and C++ code. CMC [56] is another approach to model checking C/C++ implementations, relying chiefly on hash compaction [16] and symmetry techniques to handle the large states of real programs.

Chapter 3

Design

The design of any model checker starts with several choices: whether the model checker is an explicit state or symbolic model checker, the temporal logic, and the specification language. Having addressed these issues in the previous chapter, the basic design requirements for the LF model are as follows:

- An explicit state model checker,
- using LTL as temporal logic, and
- using the LF language for specification and the compiled program as input, and
- relying in part on the existing kernel framework to support the model checker.

The main task in the design of a model checker is to select appropriate algorithms and data structures to:

- Generate and store states.
- Detect the revisiting of states.
- Perform cycle detection.

In addition to these design goals, some overriding guiding principles can be used to build an efficient tractable model checker for LF.

- Minimise the impact on the existing runtime system and compiler. Reuse, don't reinvent.
- Simple designs which can easily be debugged are preferred. Currently only a few tools are available to debug the running kernel of the LF system. As the model checker is a replacement for the kernel, the same restriction applies.
- Clear modularisation and separation of components is preferred. This allows easy extension and debugging of the model checker.

3.1 Semantic Coverage and Granularity

An important step in the design of the LF model checker is to determine which features of LF programs should be checked. The LF language is in some cases underspecified, in that it does not explicitly dictate the operational semantics, examples are:

- There is no scheduling policy is prescribed.
- Synchronisation is defined, but the selection of a partner is not,
- The SELECT feature may non-deterministically select any true WHEN clause.

To achieve an efficient, tractable implementation, the compiler and runtime system restrict the semantics by implementing a round-robin scheduling policy, a first-in-first-out pairing of communication partners and a top-to-bottom evaluation of WHEN clauses in the select.

Runtime Monitoring

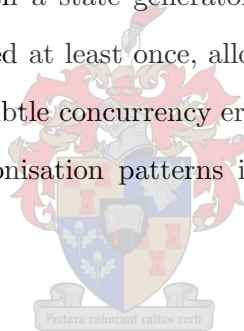
In runtime monitoring, the model checker does not direct the search. Instead, it relies on the compiler and the kernel's strict semantic interpretation to guide the execution. The model checker acts as an observer: collecting states and reporting violations. Runtime monitoring can be valuable, but it provides only a partial search of the state space. In fact, combining runtime monitoring with testing techniques may find more software defects, because a tester

can — to some extent — direct the system to explore unusual executions. One key advantage of model checking is that it explores *all* possible executions.

Abstracting Kernel Policies

In order to gain greater coverage of the semantics, it is necessary to generalise some of the decisions in the implementation of the compiler and the kernel. The compiler and the kernel conform to the semantics of LF, but they implement only of its possible interpretations. For model checking we are interested in a much broader set of interpretations — in fact, as broad as the semantics will allow. Consequently, when the model checker reports that a LF program satisfies a property, we know that the program holds independently of the scheduler and synchronisation etc. policy.

Replacing the runtime scheduler with a state generator, which ensures that each possible configuration of the program is visited at least once, allows the model checker to inspect all possible scheduling policies to find subtle concurrency errors. Extending the synchronisation policy to model all possible synchronisation patterns is merely an extension of the state generator.



Granularity

Most model checkers have a very fine granularity, typically a per-statement policy. A general-purpose model checker such as SPIN [46] has no knowledge of the target environment on which the modelled system will execute. This, in turn requires the assumption that preemptive scheduling will be used.

Modelling languages such as Promela often contain language features (for example `d_step` or `atomic`) that allow constructing atomic sequences of statements as a tool to increase the granularity. However, often the burden of deciding when these constructs are admissible is the responsibility of the user. In LF the default operation of the compiler is to build actions, large atomic sequences of statements. This approach simplifies the scheduler, and limits the state explosion when model checking. Without it, model checking LF programs would be

infeasible for all but the smallest examples.

Since states can only be collected by the model checker at the end of these actions, the model checker can only reason about the validity of the temporal properties at these predetermined points. Because the actions in LF can contain several statements, including loops and branches, tools are required to verify finer-grained correctness claims. An ASSERT statement was added to the language to solve this problem. An ASSERT accepts a Boolean expression as parameter, and may be inserted between any two statements in an LF program, without changing the meaning of the program. This can be used to state pre- and postconditions of an action. If any Boolean assertion is violated, the model checker produces a violation trace.

3.2 State Generation and Storage

At the heart of explicit state model checking is the state generator, which explores the state graph of a system on-the-fly. Depending on the requirements of the model checker a number of algorithms can be used to search through the state space: breadth-first, depth-first, or variations of these that rely on heuristics to guide the search, such as best-first. One requirement of a state generator is that eventually all possible states of the system should be visited at least once.

Typically a depth-first search is used, because it is simple to implement and satisfies the requirement of visiting each state at least once. As mentioned in 2.1.3, the result of the verification does not change if a state is visited for the second, third and so forth time. In order to avoid the redundant work, states that are encountered during a depth first search are stored. When the depth-first search *revisits* a state, it may be ignored since either the state is on the depth-first stack in which case it will be fully explored when backtracking, or all the successors of the state has been explored and no more work has to be done for the state.

Storing as many states as possible avoids redundant work, and is one of the main requirements of an explicit state model checker. The rest of this section discusses techniques to implement state storage.

3.2.1 State Representation

A state is a representation of the configuration of the system being verified at one single point in time. Each state contains information about the program counters and variables of all the components in the system.

Clearly, the more compactly states can be represented, more states can fit in the available memory. Furthermore, fewer operations are needed to manipulate a compact state. The state representation should be canonical, meaning that two distinct states should have distinct representations, and a particular state should always be represented in the same way. (This is not strictly true; we shall discuss non-injective representations in Section 3.2.3.)

The state of an LF program is in part contained by all the allocated activation records. The initial state for all LF programs is a state with only a single activation record that describes the kernel. Any action executed after this initial state transforms the state, by either adding or removing activation records, or changing the state of existing activation records.

The simplest representation is to concatenate all the activation records. However this approach has drawbacks:

- **Size.** Each activation record has at least a 72 byte header, followed by space for the local variables. As more processes are added, the size of a state quickly grows.
- **Redundancy.** The activation record contains a lot of redundant information. The redundant information is used by the normal runtime kernel for efficiency gains. Storing this information for each state is wasteful.

One of the simplest techniques to avoid storage of redundant information is byte masking [45]. A static byte mask is computed before the verification to “mask” redundant information. When constructing a state, only the unmasked information is used. The redundant information in an activation record is far out shadowed by the local variables, so this technique will not yield a sufficiently compact state.

State compaction [24] is a state compression technique that stores each variable in a model in exactly the minimum bits. This technique requires either a language which supports

enumerated types, or a model checker which can use training runs to establish the bounds on variables. Unfortunately LF does not support enumerated types, which makes implementing this technique cumbersome.

Other compression techniques such as run-length encodings, Huffman encodings and ZIP compression have been found less efficient [45] than representing each state in more complex ways. A technique that works well for LF is collapse compression described below and some semi-implicit representation techniques described in Section 3.2.2.

Collapse Compression

Instead of representing each state as a linear string of bits, each state can be represented compactly by using more elaborate data structures that mimic the underlying structure of the information that constitute a state.

Collapse compression described in [46, 45, 1] is based on the observation that the global state space of the entire system is derived from the asynchronous product of components with few local states. The premise is that if each component is represented in n bits in the state vector, of which only 2^k states ($k \ll n$) are practically reachable, $n - k$ bits are wasted in the state vector to represent the component. Instead of storing each component directly in the state vector, the 2^k values are stored in external tables, and only k bits are used in the state vector as a reference into the external tables. The external, or local state tables do require extra memory, but this extra cost is offset by the total savings incurred by not storing $n - k$ redundant bits per component per state.

The LF system lends itself well to adopting this scheme as activation records present a clear separation of the components, and often processes contain few local states. Aside from these properties, it is clear that an activation record contains a large amount of redundant bits. Consider, for example, a single field in the activation record, the next action pointer. Instead of using a table with start addresses of actions and only storing an index into this table, the start addresses of actions are stored directly in this field. This requires 32 bits even though most LF process definitions have fewer than a hundred actions. Similar observations can be

made of other fields in an activation record.

Collapse compression compactly represents LF programs in only a few bytes per state, with only a minimal contribution of the local state tables to the memory requirements of the state store.

3.2.2 State Storage

A compact representation for each state is only one half of the problem. Storing states to ensure fast and reliable lookup and insertion of states into the state store completes the picture. To this end the requirements for state storage are:

- As many states as possible should be stored.
- State insertion and lookup should be fast.
- No state should leave the store before the model checker has completely explored that state.

Hashing

Most explicit state model checkers use hash tables to store the state space. To ensure the most even distribution of states in the hash table, the hash function should take as many of the bits of each state as possible into account. There are two approaches to deal with collisions: chaining and rehashing.

Hashing with chaining stores all the states that hash to the same slot in a secondary data structure such as a linked list. Each state that hashes to that slot is placed in the data structure, and the access time is related to the size of the data structure. In the case where the secondary data structure is a linked list, the overhead of the pointers used to link the elements together can be large if the states are small.

Closed hashing stores states directly in the hash table. This avoids the overhead of the secondary data structure. A drawback of this approach is that each state must have the same

size. In systems which support dynamic process creation this presents a problem since an upper bound on the number of processes is required. It is not unreasonable to expect the user to supply such an upper bound. If the upper bound is too loose this translates to more waste per state, but fixed sized states simplify memory management.

For the sake of simplicity, the LF model checker relies on closed hashing with two independent hash functions and quadratic probing to avoid clustering. Furthermore, if collapse compression is used, the global states of LF programs will be relatively small, and the overhead of the secondary data structure in hashing with chaining may be significant.

Implicit Storage

Explicit state storage techniques rely on associating a fixed set of bits with each state. Implicit storage techniques represent the entire state space in a compact way by exploiting the internal structure of the state space. The graph-based techniques, such as sharing trees [33], minimised automata [42] and ordered binary decision diagrams [1] are the most well-known implicit techniques. They can achieve up to a factor 17 reduction in memory requirements, at an up to ten-fold increase in running time [42]. Because violations of temporal properties are generally found early during the verification these techniques are best used as an extension of the normal operating mode of the model checker. Although these techniques perform well, they are not immune to an exponential blow-up in memory requirements, and the implementation is slightly more complicated than hashing.

Other semi-explicit techniques such as difference compression [60] and Δ -markings [66], store a subset of the entire state space explicitly, and the remainder implicitly. The basic principle of these two techniques is that each transition in a state graph only changes a small portion of the state. The difference compression technique stores an arbitrary subset of the state space explicitly, and the remainder of states are stored as the difference in bits between a new state and an explicitly stored state. The Δ -markings method encodes a subset of the states explicitly, and stores a sequence of action addresses for the remainder. The implicitly coded states can then be reconstituted by executing the sequence of actions.

In conjunction with the collapse compression used in our model checker, the Δ -markings method may yield good reduction in storage requirements, and can be investigated as an extension to the model checker at some future time.

3.2.3 Storing Large State Spaces

The number of states quickly grow as the complexity of the model grows. Often the best compression techniques still do not incur a great enough saving to complete the verification due to a lack of memory to store the reachable states. Several techniques have been proposed to extend the state storage capabilities:

- State caching
- Selective state storage
- Bitstate hashing and similar methods

State Caching

State caching is a very popular technique used in explicit state model checkers. During a depth-first search of the state space only the states on the current execution path need to be remembered to ensure termination. States on other previously explored paths are stored merely to avoid redundant work. Since there is no more work to be done for fully explored, or backtracked states, replacing them with newer states does not invalidate the search, but does commit the model checker to re-explore the state, should it be revisited.

Selecting appropriate candidate states for replacement were studied in [41]. These strategies include replacing the most frequently visited states, least frequently visited states, random states from the largest class (classes are formed by states visited equally often), and a round-robin replacement. The results of the experiments show that there is no clear correlation between how frequently a state is visited and the probability of revisiting a state. The round-robin replacement strategy was chosen as the best bargain, striking a balance between the cost of finding a candidate for replacement and the cost of double work. In [30] it is shown that

the performance of state caching can be improved if combined with partial order reduction techniques.

The theoretical lower bound for state caching is using a cache exactly the size of the longest simple depth-first path in the model. However, in practice most tests show that state spaces of roughly double or triple the size of the state cache can be explored. The limiting factor in using state caching is the amount of double work required as states are replaced in the state store.

The amount of double work is limited by how many successors of a state are still in the cache. As the number of replaced states increase, the probability that all the successors of a state is still in the cache decreases. Generating the successors will then again replace some other states, which may again cause some double work for other paths. In the worst case, a cascade of replacements may happen. In this case, each new state replaces a successor of itself, which will again replace another state.

One approach to avoiding the cascading effect is stratified caching [23], which limits the amount of double work by dividing the state store into different strata. Each stratum contains states at the same depth modulo some constant of the depth-first search path. Only replacing states from one stratum at a time, ensures that the successors of the replaced state has a high probability of still being in the state cache.

Selective Storage

A dual of state caching is selective storage of states. The technique in [2] relies on the observation that many models and protocols often exhibit long chains of states with only a single successor. Storing only a single state as a representative of such a chain of states should suffice. The goal of the technique is to represent each cycle in the system with a single state. However because it is very difficult to predict whether a state is a good representative of a cycle, every k 'th state is stored. The premise is that the runtime cost of regenerating the states leading to the single state stored for a chain or cycles, is far outweighed by the memory requirements of storing all the states. The authors further elaborated on using static analysis

and heuristics in determining which states are likely to form a covering set, that includes a state from each strongly connected component in the state graph.

Another selective storage technique, the sweep line method from [6] relies on a technique similar to garbage collection. It uses a notion of progress in state spaces to identify states that will never be reached again and removes these states from the state store. Unfortunately the technique relies on a progress measure function that the user has to supply, and this is often hard to formulate.

Lossy Storage

Lossy storage techniques rely on mapping each state non-injectively onto an arbitrarily small number of bits before inserting it in the state store. This improves the memory requirements of the state store, but at a price. Two distinct states can share the same representation, and therefore if one state is already present in the state store, the second may be mistakenly recognised as a revisit. This causes a portion of the state space to be ignored by the model checker. This is a serious drawback as an error may be missed by the truncated search and therefore cannot guarantee that a property holds for a model. Fortunately, any error that is reported during such a truncated search, is a real error.

One example of lossy storage is bitstate hashing [43]. A large fixed sized bit array is used for state storage. A hash function is used to compute an index for each new state, and the corresponding bit is set to indicate the state has been seen. The main premise is that if the bit array is chosen large enough the probability of collisions is very small. To further reduce the probability of collisions (and therefore omission) it was suggested in [44] that two bit arrays, with statistically independent hash functions, are used.

Bitstate hashing is the most extreme example of lossy storage. The lossy compression scheme in [79] ameliorate the omission probability by using several bits as a highly compacted version of the state vector. This compacted state vector is computed by a hash function. The compacted states are again stored in a large hash table, with collision avoidance, to further reduce the omission probability. In [17] the scheme is combined with state caching in order

to further increase the state storage capability.

3.3 Temporal Properties

Safety properties of the form $\Box p$ where p is an atomic proposition, are invariants of the system, detecting violations these properties are equivalent to reachability. Liveness properties express that some desirable state is eventually reached. Examples of liveness properties are fairness and responsiveness. Detecting violations of LTL properties is equivalent to finding a cycle with one more accepting states.

Properties as Processes

LTL properties are often expressed as Büchi [76] automata for model checking. To accommodate Büchi automata the LF language was extended to express properties as special processes. LF property processes are indicated in the program by declaring it a PROPERTY instead of a PROCESS. If a property process is present in a module, it will be transparently created when the module is initialised. A similar method is used in the SPIN model checker to specify properties, namely the **never** claims.

In short, a property process may contain local variables, any LF statements, and does not have any parameters. Furthermore two intrinsic functions are needed to accommodate Büchi automata: ACCEPT and NOMOVE. The ACCEPT statement signals to the model checker that the property process has entered an accepting state. The NOMOVE statement signals to the model checker that there is no viable transition of the automaton for the synchronous product.

The Büchi automata used in model checking are usually non-deterministic, and LF does not explicitly support non-determinism. To remedy this problem, a non-deterministic operator CHOICE was added to the language. The semantics of this operator is that it allows non-deterministic selection of actions when model checking. This feature is discussed in Section 4.2.2. For the purpose of this discussion it is only necessary to note that the choice operator

```

1  PROPERTY GFp;
2  VAR   s : INTEGER;
3  BEGIN
4      WHILE TRUE DO
5          IF s = 0 THEN
6              CHOICE
7                  s := 0
8              OR
9                  IF (p) THEN
10                     s := 1;
11                     ACCEPT
12                 ELSE
13                     s := 0
14                 END
15             END
16         ELSIF s = 1 THEN
17             RESCHEDULE;
18             s := 0
19         END
20     END
21 END P;

```

Figure 7: Property process for $\Box\Diamond p$, where p in the example is a Boolean statement

causes a reschedule before non-deterministically selecting an action.

An example of an LF property process for $\Box\Diamond p$ is shown in Figure 7. Property process GFp has two states and three transitions that correspond exactly to the states and transitions of the Büchi automaton in Figure 3. Instead of executing a property process as usual, the model checker schedule one action of the property for each action that can be executed from the current state.

The next two sections discuss two approaches to detecting accepting cycles.

3.3.1 Nested Depth-First Searches

The nested depth-first search (or NDFS for short) originally proposed by Coucoubertis et al. [12], performs a first depth-first search to: (1) find and (2) sort accepting states according to the last visit. When a state has been fully explored and it is accepting, a second depth-first is started to determine if the state is reachable from itself. An outline of the NDFS algorithm

```

1  PROCEDURE CHECK;
2  BEGIN
3      DFS(initialstate)
4  END;
5
6  PROCEDURE DFS(source);
7  BEGIN
8      IF marked0(source) THEN RETURN END;
9      mark0(source)
10     FOR ALL t = SUCCESOR(source) DO
11         DFS(t)
12     END;
13     IF ACCEPTING(source) THEN
14         NDFS(source,source)
15     END
16 END;
17
18 PROCEDURE NDFS(source,seed);
19 BEGIN
20     IF marked1(source) THEN RETURN END;
21     mark1(source);
22     FOR ALL t = SUCCESOR(source) DO
23         IF t = seed THEN
24             REPORT cycle
25         ELSE
26             NDFS(t,seed)
27         END
28     END
29 END;

```

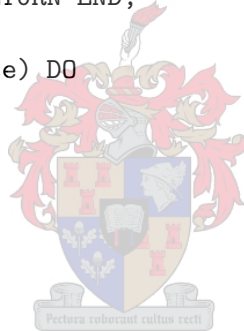


Figure 8: The Nested Depth-First Search Algorithm

is shown in Figure 8.

The NDFS algorithm can be improved to find accepting cycles earlier in some cases. One case is when a state is found during the second depth-first search which is already on the depth-first stack of the first search. Since we already know there is a path from this state to the accepting state, no further work has to be done to report a cycle. This is the basis of the improved algorithm suggested in [40] and also the algorithm used in the SPIN model checker.

A second improvement to the algorithm was suggested in [22]. If the current state is accepting and a successor of the state is on the depth-first stack, there is no need to start a second

depth-first search.

A further improvement of the original algorithm that combines the improvements from [40] and [22] is presented in [67]. Furthermore, [67] survey the existing cycle detection algorithms, both NDFS based and algorithms based on finding SCCs (Which we describe below).

Nested depth-first searches have several drawbacks. Chief among these is the fact that each state may be visited exactly twice if there is no cycle that contain the accepting state, doubling the runtime requirements. Since cycles are only detected while backtracking, many more states than is necessary to illustrate the violation will be explored. Determining the cycle again rely on a depth-first and the violation cycle may not be the shortest cycle containing the accepting state. In [22] the authors proposed a strategy that can produce the shortest cycles that contain an accepting state, but it requires possibly visiting each state several times.

3.3.2 Strongly Connected Components

Another approach to finding accepting cycles is to detect strongly connected components (SCC) with one or more accepting states. An SCC is a maximal set of states S such that for any two states $s, r \in S$ there is a path from s to r . Two widely known algorithms for detecting SCCs are Tarjan's algorithm [71] and Kosaraju's algorithm [11, 68]. Both algorithms use a depth-first traversal of the state graph. However, the Kosajaru algorithm requires a second depth-first traversal of the transpose of the state graph.

In order to avoid the generation and storage of the entire state graph, for the Kosajaru algorithm to work, SCC-based model checkers rely on Tarjan's algorithm. The algorithm requires two stacks: the depth-first stack, and a component stack, which records partially explored SCCs. In addition to the second stack, the algorithm requires associating two values with each state: the depth-first number, and the low-link value. During the depth-first search new states are placed on both the depth-first and component stacks. A state remains on the component stack until the entire SCC has been explored. When the depth-first search backtracks past a state and it is the root of an SCC, all the states of the SCC are removed from the component stack.

The low-link value associated with a state s is the lowest depth-first number of any state that can be reached from s . As proved in [71], a state forms the root of an SCC if and only if its depth-first number is equal to its low-link value. This is how the algorithm recognises SCCs. Tarjan’s algorithm reports SCCs as soon as all the states that form an SCC have been explored, and, similarly to the NDFS approach, starts with the “deepest” SCCs first. However, in contrast to the NDFS approach, only a single pass over the state space is necessary to detect cycles. Tarjan’s algorithm reports a violation as soon as all the transitions of an accepting cycle have been explored. This early detection is desirable because it reduces the memory requirements and improves time consumption of the model checker.

The extra memory overhead of Tarjan’s algorithm is usually cited as argument for NDFS [12, 29]. However, [25] and [26] argues that Tarjan’s algorithm can be implemented efficiently, and notes that combining the state space with a Büchi automaton often breaks the system into many smaller SCCs, which limits the storage requirements of the algorithm.

The TCHECK algorithm in [26] rely on quickly searching the component stack to detect when a cycle that contains accepting states is closed. The algorithm differs from the original Tarjan implementation in that it is iterative and dispenses with the need to store depth-first numbers. Although it uses three stacks, the space for the stacks can be minimised by immediately inserting the state into the state store and only recording a reference to the state in the stacks. The main argument in favour of using the TCHECK algorithm of [26] in our model checker can be summarised as follows:

- It has a clear iterative presentation, which is well suited for LF and it eliminates the use of kernel stack space.
- Even at the expense of extra space for stacks, the algorithm generally produces shorter counterexamples than NDFS, which is a great aid in understanding the nature of a violation.
- The states of an LF program is large, and doubling the number of states that need to be explored may cripple the model checker more than an efficient stack representation.

- Because the algorithm already detects SCCs, a slight modification of the algorithm can introduce strong fairness.
- An initial version of the LF model checker relied on Tarjan’s algorithm for cycle detection.

Another cycle detection algorithm based on detecting SCCs was introduced in [13] and is based on ideas from the SCC detection algorithm described in [15]. Like the algorithm [25] only a single depth-first search is necessary.

3.4 Closing Open Systems

A reactive program is considered open when there is an interaction with some external environment. Typically in model checking only closed systems are considered. This implies that before an “open” system can be verified, some representation of the environment needs to be constructed to “close” the program. This process usually involves manually constructing an abstract representation of the external environment. Providing a realistic model of the external environment is a difficult and time consuming task.

The LF kernel only provides the most necessary device drivers to support the execution of a program. Instead of implementing device drivers in the kernel space, the LF language and runtime system provide support to develop user-level device drivers. It is a reasonable assumption then that a developer might require facilities to model check device driver implementations. As it is often very difficult, or even impossible to determine (and restore) the state of a hardware device, the LF system must provide some means of producing a suitable environment to enable the development of hardware drivers.

3.4.1 Manual Methods

As stated earlier, manually deriving a model of the external environment is difficult. One approach is by refinement of a universal environment. A universal environment is an abstract model of the external environment, that can at any time provide any signal or result to a

program. Typically for LF programs this amounts to an LF process which can send and receive any message type on any channel that communicates with the environment. This environment can then model any behaviour that an LF program may require from the external environment. However this representation of the environment is too abstract, and may cause behaviour that is not encountered in the real environment. Furthermore, it may lead to an unacceptable increase in the size of the state space.

The problems with the universal environment can be overcome by using refinement. Using the execution traces of unrealistic behaviour the universal environment can be refined to closely mimic the underlying device. In some cases this can increase and in other cases this can decrease the number of states. The refinement should strike a balance between a realistic model of the environment and the number of states generated.

To support the development of user-level device drivers, the LF language provides routines to access memory-mapped I/O ports on the target hardware through the `PORTIN` and `PORTOUT` intrinsic functions. It also provides a module `SYSTEM` which gives the user access to the interrupt channels of the target hardware. Furthermore, the LF language allows the user to define static addresses for variables. For model checking the code of the LF program should be modified to map these I/O ports and variables to communication with an environment process.

An example of how a general device environment can be modelled in LF is seen in Figure 9. Each process in the *GenDevice* module represents a hardware component. The internal state of the device can be driven by non-deterministic choice through the `CHOICE` operator, discussed in Section 4.2.2. To mimic the underlying hardware, the general device framework should be composed with the original device driver to yield a closed reactive system.

3.4.2 Automatically Deriving Environments

A universal environment can also be extracted directly from the source code. This entails identifying the interface the software system uses to communicate external devices, and creating a general environment, or replacing the control structures which determine the reaction

```
1  MODULE GenDevice;
2  IMPORT SYSTEM;
3
4  PROCESS Device(int : InterruptChannel);
5  CONST DevicePort1 = 2FFH
6  VAR notdone : BOOLEAN;
7      ins : INT8;  msg32 : INT32;
8  BEGIN
9      WHILE TRUE DO
10         int!INTERRUPT;
11         notdone := true;
12         WHILE notdone DO
13             SELECT
14                 WHEN SYSTEM.IOPORT?PORTIN(ins,portno,msg32)
15                     & (portno = DevicePort1) THEN
16                     (* change state of device *)
17                     notdone := false
18                 WHEN SYSTEM.IOPORT?PORTOUT(ins,DevicePort1,msg32) THEN
19                     (* change state of device *)
20                     notdone := false
21             END
22         END
23     END
24 END Device;
25
26 BEGIN
27     NEW(SYSTEM.InterruptN);
28     CREATE Device(SYSTEM.InterruptN);
29 END;
```

Figure 9: A general device model

of the system with the external environment by non-deterministic choices.

The technique discussed in [9] would be invaluable in LF, especially for the development of device drivers. Instead of computing a general environment directly, static analysis is applied to determine which variables or procedures interact with the environment. Where a conditional statement depends on variables or procedures that interact with the environment, it is replaced by a non-deterministic choice. This approach eliminates simulating the environment with a separate process, which avoids the overhead of evaluating more processes during model checking. The technique does however over-approximate the behaviour and may introduce executions that may not be feasible in reality, but the process is automated and can close any open software system.

Implementing this technique in the current LF compiler will be an arduous task as the compiler does not generate any of the required data structures, such as dependence graphs for variables. The new compiler, still under construction, does include these and other data structures to accommodate program slicing. A worthwhile experiment may be to integrate this functionality into the LF slicer [51].

3.5 Model Checking Embedded Applications

One of the most serious constraints in embedded software is the limited resources such as memory and processing power. Especially the memory limitations hamper building a model checker for embedded applications.

Techniques such as bitstate hashing may be useful in such memory-constrained systems. Bitstate hashing can compactly represent those states that have been fully explored. Still it cannot be used to as the only record of the states on the depth-first stack. More complete information is needed about these states to generate successors. This can be expensive because of LF's large states, but other techniques, (e.g., collapse compression) can help. Another approach to the problem is to only apply the state space exploration techniques to portions of the program that the user feel may contain or cause errors. In this case a small state cache can be used, since only a fraction of the potential state space will be expanded.

The limitations imposed by the target hardware are far from easy to overcome. One possibility is to use a special version of the hardware with significantly more memory. This may solve the storage problems, but it fails to address the question of processing power which is much more closely tied to a specific architecture. Alternatively, the system can be model checked on another architecture altogether. In this case, it is reasonable to expect the same program to behave similarly, given correct code generation and a direct correlation between the two execution environments. Small errors may be introduced when moving between two architectures, but a correct compiler implementation would not affect the control flow of a program.

The current implementation of LF is aimed at the Intel 386 family of processors, that are currently employed in many embedded devices. As a result LF programs can be developed and tested on widely available desktop hardware. This influences the model checker design in that, instead of using non-injective state storage techniques that trade accuracy for memory consumption, our model checker can use techniques such as collapse compression. Since the LF model checker will be used during development, the expectation is high that a property may be violated, and it is important to illustrate these cases quickly.

Conclusion

In short, the the LF model checker relies on the collapse compression scheme to represent the global states. It uses closed hashing with rehashing and state caching for state storage, and a variation of the Tarjan algorithm for cycle detection.

Chapter 4

Implementation

The runtime kernel and compiler were mostly complete when work on the initial version of the model checker was started. This placed several constraints on the model checker, in particular:

- The code generated by the existing compiler should be used.
- The model checker should fit seamlessly into the existing runtime system.
- The model checker should support the LF language semantics, which is not necessarily completely supported by the existing compiler and runtime system.

In the previous chapter several techniques were discussed to implement the components of an explicit state model checker. The following techniques were selected to implement the LF model checker:

- The improved version of Tarjan's algorithm is used for cycle detection [26].
- Collapse compression is used to represent states [1].
- Closed hashing with state caching is used for state storage.

This chapter covers some of the implementation details, specifically state generation and detection of visited states.

4.1 System Organisation

The LF compiler and runtime system is implemented in Oberon [57] and the model checker follows suit. The LF model checker comprises roughly 1500 lines of code and is linked into the runtime system. The model checker serves as a replacement for the normal LF kernel, and shares a lot of the original kernel implementation. Initially, it was conceived that the model checker should only be an extension to the original LF kernel that interacts with it through the scheduler interface. However, this approach soon turned out to be insufficient. The message passing system in particular required more and more “hooks” into the kernel to extract a clear picture of the system at one time. Also new system calls were added to accommodate features such as assertion checking, non-determinism, and properties.

Memory Organisation

Figure 10 depicts the current memory layout of the model checking kernel. The lower 65 KB of the memory is allocated for the boot table and the kernel execution stack (Here and in the figure KB denotes 1024 bytes, and MB denotes 1024 KB). To accommodate the model checker code, the space allocated for the kernel is increased from 64 KB to 128 KB, and the LF program heap is moved from 129 KB to 193 KB. All the memory from 193 KB up to 1 MB is used for the LF program and activation record heap.

The activation record heap is divided into two sections: the pool allocation table, which records the current allocation status for each process type, and the activation records themselves. An entry in the pool table is the root of a linked list. Whenever a process is allocated and the linked list associated with the process type is empty, an activation record is allocated from the heap. When the process terminates, instead of returning the activation record to the heap, it is inserted in the linked list of the process type. When a process with the same type is reallocated, the first entry in the pool table is used, rather than allocating from the heap.

All the memory above 1 megabyte is used for the main model checker data structures. This memory is divided into two sections, space for the depth-first search stack and cycle detection

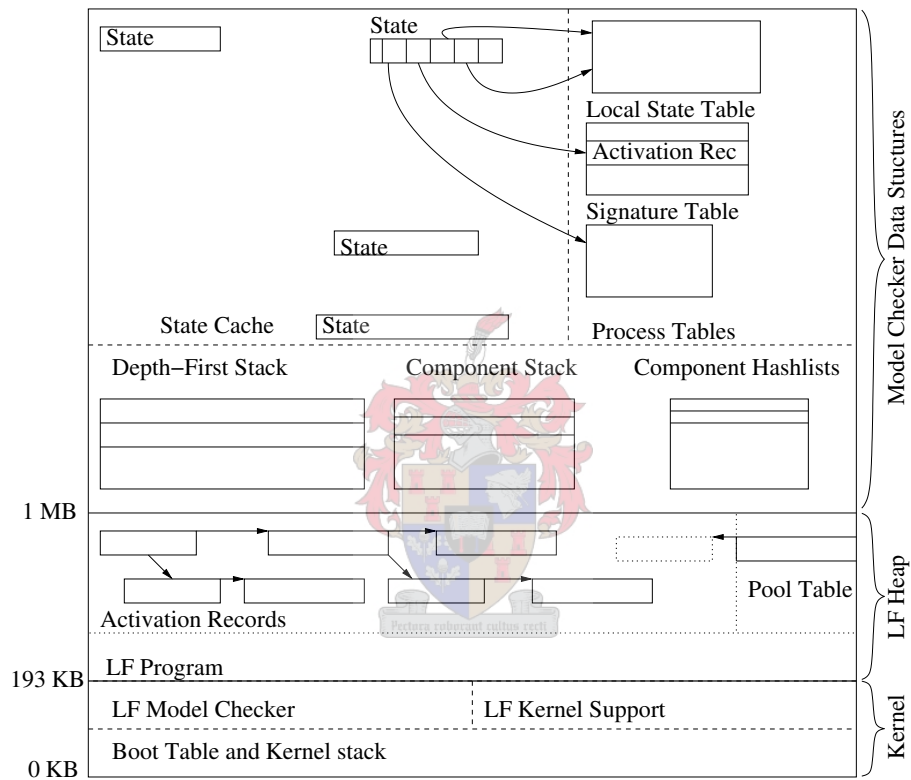


Figure 10: Memory Organisation of LF Model Checker

data structures, and the state cache.

The depth-first stack and cycle detection data structures are allocated when the model checker is initialised. The size of the two stacks and the hash lists are described by a parameter in the model checker. When the stack entries are exhausted the search is terminated with an error message that advises the user to increase the stack size.

The largest portion of the available memory is allocated for state storage. State storage comprises two parts, storage for the local states and, a large hash table to store the global states. One local state table is allocated per process type during the initialisation of the state cache. All the remaining memory is allocated for the state cache.

One invariant that has to be maintained is that the information contained in either the state cache or the depth-first stack should be sufficient to completely restore the LF heap to a previous configuration. To keep the memory contribution of the depth-first stack to a minimum, states along the current depth-first path are immediately inserted into the state cache, and only a reference to the state is kept on the stack.

4.1.1 Module Layout

Because the LF model checker and original kernel implementation are intertwined, a revised kernel module layout is shown in Figure 11. This should be compared to Figure 6. As we can see, the organisation of the kernel is more or less the same. One significant difference is the replacement of the scheduler with the depth-first search module, and the addition of the state cache.

The kernel drivers enable the system to access devices such as the interrupt controller and timers. The `LFBoottable` module is used during the initialisation of the kernel to load the drivers required by the kernel and to set the options of the model checker. The system calls of the original kernel are mostly intact in the model checking kernel. The exception is the communication system calls which were extended to signal the model checker when a channel is accessed.

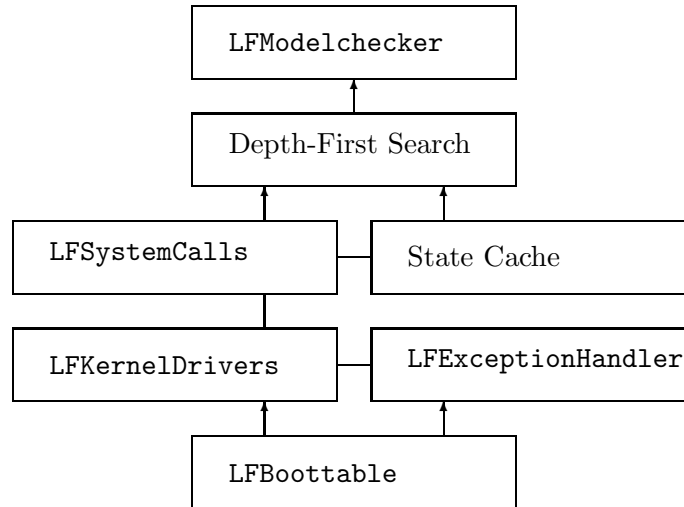


Figure 11: The module layout of the model checking kernel

4.1.2 Additional System Calls

In order to support the new features required by the model checker (non-determinism, property processes and assertion checks), extra system calls were added to the runtime system. Specifically five new system calls were added:

AssertCall halts the system if the value of the boolean expression is false. When the model checking kernel is used, this produces an execution trace to the violation.

ChoiceCall loads the address of a table with the non-deterministic choices into the activation record.

CreateProperty instantiates a property process.

AcceptingState notifies the model checker that the property process has entered an accepting state.

NoProgressCall notifies the model checker that there are no viable actions from the current state in the property process.

4.1.3 Defining the System State

In addition to the active processes, there are several other data structures in the LF kernel to consider when forming the state: data structures used for heap management, process accounting variables, channel records.

To improve its efficiency and reduce its memory requirements, the LF kernel avoids the use of a procedural stack. Entry into the system calls are through jump instructions, rather than the normal enter and exit calls which the Intel instruction set provides. The kernel procedural stack is only retained to accommodate the model checker. Most of the LF kernel is written in the Intel 386 assembler language. In contrast to this, the model checker is written in Oberon, which require the procedural stack. At the end of an action, when new states are collected, the kernel stack is empty, and therefore need not form part of the state.

The heap management data structure, the pool table should form part of the state. Consider, for instance, the following scenario: if process A was created and terminated along the current search path, the pool table will record that the pool now contains the discarded activation record of A. If the model checker backtracks to a point where process A is still active, and another instance of the same process type is created, the kernel memory management will mistakenly allocate the activation record of A to the new process. To skirt this issue, the pool table and process accounting form part of the activation record allocated for the kernel. To avoid storing all the possible free activation records, only the initial pool entry is kept in the pool table, this does lead to some fragmentation in the heap, but it has an acceptably small effect on the execution time of the system, and other than a normal execution, the model checker will terminate.

In addition to the pool table, there is still one data structure which needs to be considered when forming a new state of the system, the channel records.

Interprocess Communication System

In concurrent languages processes may communicate either by shared variables or message passing. Shared variables are very simple to implement and fast since only a single assignment

is required to disseminate a value. On the other hand exclusive access to shared variables may sometimes require the use of complex locking mechanisms. Message passing is a more convenient way of modelling the interaction between processes. LF supports synchronous message passing.

In a model checker which supports synchronous message passing, it is usually not necessary to store data structures such as the channel records. The information necessary to synchronise two processes on a channel is implicitly coded in the program counters of the processes. If a process wishes to synchronise with another process over a channel it is only necessary to scan the processes until one is found which wishes to execute a complementary action on the same channel.

To avoid traversing all the blocked processes each time a communication statement is executed the kernel provide a summary of which processes are willing to communicate over a channel in the channel record. The message types which may be transmitted over the channel is each represented in a channel record by a *symbol record* that contain the head and tail of three queues. One queue for senders, receivers, and selective communication. An process may only be in one queue at a time.

Whenever a communication statement is executed by an processes, and there is no process waiting to execute an complementary statement, the process is blocked and placed at the back of the applicable queue. If the complementary communication statement is later executed, the kernel removes and unblocks the head of the relevant queue and synchronise the two processes.

This is an important optimisation in the kernel implementation. Removing the channel records would require a partial redesign of the message passing infrastructure in both the kernel and the compiler. Furthermore, retaining the channel records will have same efficiency benefits, that it provides the kernel, while model checking.

The repercussion of this is that the channel records should form part of the state of the system. This has two effects on the model checker, firstly the state generator throughput is reduced since states are larger. More importantly, since the channel records explicitly record

```

1  TYPE  StackEntry = RECORD
2      State : POINTER TO SC.CompressedState;
3      Last  : ActivationRecord;
4      PartnersLeft,Choices,PropertyChoices : SHORTINT;
5  END;
```

Figure 12: The Depth-First Stack

information about the queues this must lead to extra states.

4.2 Generating States

The state space is generated on-the-fly using depth-first exploration. To generate states the following information is needed:

- the current state,
- the states on the depth-first stack,
- the set of actions which has to be explored for the current state, and
- the set of fully explored states.



The current state of the LF system is kept both in compressed form that is inserted in the state cache, and an uncompressed form on the LF heap. An element of the depth-first stack consists of two parts; the state, and the scheduling information. Instead of duplicating all the entire scheduling queue on the stack, only the most pertinent information required to rebuild the scheduling queue is stored on the stack. The scheduling information contain: which process and action executed was last explored, the number of non-deterministic choices left, and the remaining synchronisation patterns for the last action. This information is used while scanning through the uncompressed current state to select a process and action to generate a new state. Figure 12 illustrates a stack entry.

In addition to the depth-first stack, two extra data structures are managed by the depth-first search engine, namely the component stack and component hash lists. The component stack

stores the components of partially explored SCCs and the hash lists are used to quickly search through the component stack. For the details see [26] and Section 3.3.2.

The depth-first stack is operated accessed through `Push` and `Pop` operators. `Push` attempts to insert a new state into the state cache. If the state has not been visited before it is inserted in the state cache, and a reference to the state is stored on the stack. If this state has been visited before, no extra work is needed since the state is either fully explored, or is still on the depth-first stack. In this case, the topmost state on the depth-first stack is restored to the heap. The `Push` operation has two possible side effects: either the state is a new state and added to the depth-first and component stacks, or if it is a revisited state and it closes an accepting cycle, in which case a violation is reported.

The `Pop` procedure removes the topmost state from the depth-first stack. If the last state has been popped off the stack, the search is terminated. Otherwise, the heap is restored to the configuration of the new topmost state on the depth-first stack. The search continues with this state.

4.2.1 Depth-First Search

The depth-first search engine is the heart of the model checker, and is the main interface between the model checker and the rest of the system. The search engine replaces the usual *Reschedule* system call in the LF kernel, and dispatches a new process and action to the kernel whenever a reschedule call is invoked.

After the runtime system is loaded into memory by the boot loader and all the kernel setup is completed, the runtime initialises the LF program by executing the entry code of the LF program. This creates an activation record to represent the program and initialisation code, and transfers control to the scheduler through a system call. At this stage there are only two activation records in the system, the activation record that represents the runtime and the activation record of the LF program. This is the first state of the LF program that is pushed onto the depth-first stack. This state has only a single action which can be explored, the initialisation code of the program, which creates the activation records for the modules and

the property process.

After the initialisation code of the program has been executed, the depth-first search engine alternates between two modes. It either schedules an action from the property process, or selects a regular process to continue construction of the asynchronous product. A Boolean flag is used to indicate which mode is active.

The main task of the depth-first search is to select processes to execute actions in order to generate all the possible states. The rest of this section will briefly look at how processes are selected to generate the state.

4.2.2 Selecting Actions

After the execution of an action from the property process, the depth-first search attempts to select a new process and action to continue the search. If the last state inserted into the state cache is a new state, the Push operation initialises the topmost stack entry to indicate that no successors has been generated for the state.

If the new state has no enabled process the state is deadlocked, in which case either an error trace is produced by the model checker, or the model checker can ignore the state and backtrack. If the state is not deadlocked the enabled process with the lowest process identifier is selected and executed. The function *FirstEnabled* in Figure 13 finds the enabled process with the lowest process identifier, and sets the *Last* field of the topmost stack entry to indicate the last process explored.

Once a process is selected, the reschedule system call initialises the machine registers for execution of a process, loads the address of the activation record into a register for address resolution of local variables, and loads the address of the next action into the instruction register of the processor.

If the newly generated state has been encountered before, the state is not pushed onto the stack and topmost stack entry is used to select a new process to continue the search. In this instance, selecting a process to continue the search is more complex. There are four distinct cases to consider:

```

1  PROCEDURE Reschedule;
2  BEGIN
3    IF Property THEN
4      result = Push();
5      IF DFStack[TopState].PropertyChoices # 0 THEN
6        SetAction(PropertyAR, );
7        DEC(DFStack[TopState].PropertyChoices)
8      END;
9      NextAR := PropertyAR;
10     Property := FALSE
11   ELSE
12     IF result = NEW THEN
13       IF Enabled(DFStack[TopState].State) = 0 THEN
14         HALT(DEADLOCK);
15       ELSE
16         NextAR := FirstEnabled(DFStack[TopState].State);
17         INC(DFStack[TopState].Explored)
18       END
19     ELSE
20       WHILE DFStack[TopState].Enabled - DFStack[TopState].Explored = 0 DO
21         Pop()
22       END;
23       Restore(DFStack[TopState]);
24       IF DFStack[TopState].Partners # 0 THEN
25         NextAR := LastExplored(DFStack[TopState]);
26         DEC(DFStack[TopState].Partners)
27       ELSIF DFStack[TopState].NDChoices # 0 THEN
28         NextAR := LastExplored(DFStack[TopState]);
29         SetAction(NextAR, );
30         DEC(DFStack[TopState].NDChoices)
31       ELSE
32         NextAR := NextEnabled(DFStack[TopState]);
33         INC(DFStack[TopState].Explored)
34       END
35     END;
36     Property := TRUE
37   END;
38   SetUpRegisters(NextAR);
39   JUMP(NextAR.Action)
40 END Reschedule;

```

Figure 13: The Depth-First Search Engine

- No more successors can be generated, all the enabled processes have been explored. In this case, the topmost state of the depth-first stack is removed from the stack and the new topmost state is restored, and selection of a suitable candidate can continue.
- The last process that executed for the topmost state either ended in no communication statement, or all the communication partners have been explored, or all the non-deterministic choices have been explored. In this case the next enabled process is selected.
- The last action executed ended in a communication statement with multiple partners that still have to be explored.
- The last action executed was a non-deterministic choice and there are more choices left to explore.

We will elaborate on the last two cases below.

Communication

The synchronisation policy of the the kernel is generalised by investigating every possible pairing of communication partners. Whenever a process executes an communication statement, the model checker first pairs it with the head of the queue. The *PartnersLeft* field of the topmost stack entry is set to the length of the rest of the queue. When backtracking to a state with a non-zero *PartnersLeft* field, the model checker knows there are still other blocked processes to be considered and will explore the next pairing. After each such step the *PartnersLeft* field is decremented.

In the original runtime implementation, the blocked processes are stored in a first-in-first-out order. If, for instance, there are four processes each which wish to communicate over a single channel, the order in which the processes are explored may lead up to $4! = 24$ possible permutations of the processes in the queue. Since the order in which processes are paired in the kernel is abstracted in the model checker, storing the all the possible permutations of the queue entries is redundant.

```
ChoiceOp := "CHOICE" Statementblock {"OR" Statementblock} "END"
Statementblock := Statement {";" Statement}
```

Figure 14: The BNF of the CHOICE operator

To avoid the overhead of storing the all the possible orderings, the model checking kernel organises the queue of blocked processes as an ordered list with an insertion sort. The processes are organised on the addresses of the activation records. Although the insertion sort incurs an extra runtime penalty when no communication is feasible, the possible saving in states justifies it.

Non-Determinism

The LF language is designed to be an implementation language and therefore did not originally support non-determinism. As described in Section 3.4, non-determinism is a crucial ingredient in deriving closed systems. The *CHOICE* operator was added to language to solve this problem. The operator allows modelling a non-deterministic choice of an action when model checking. The BNF of the feature is shown in Figure 14.

Each statement block is compiled into an action as usual, and the start addresses of the choices are stored in a table in the executable. An extra system call was added to the kernel to load the address of the choice table into a field of the activation record, and reschedule.

When a new process is selected to execute and the *NDChoice* field of the activation record is set, the model checker selects the first action in the table of non-deterministic actions to execute. The number of remaining options is loaded into the the *ChoicesLeft* field of the topmost state. The *Choices* field in the stack serves as an index into the table of non-deterministic choices.

When backtracking to a state with a non-zero *ChoicesLeft* field, the same process that was last explored is explored again. Before executing an action for this process the first of the remaining choices is selected and the address of the action is loaded into the *Action* field of the activation record of the process to explore. Only when the *Choicesleft* field is zero will the next enabled process be evaluated. This ensures that all of the non-deterministic choices

are evaluated.

The same procedure is applied for the property process, except that the *PropertyChoices* field is considered.

4.3 State Storage

The main data structure in a model checker is the state store, which records the states that have been visited during the verification. Our model checker implements state storage using the collapse compression to represent each state in the state store, and uses state caching to allow the model checker to explore more states than the physical memory allows.

4.3.1 Local States

The local state of a process is entirely contained in the activation record, and the state of a channel in the channel record. Activation and channel records are allocated either from the heap, or from the pool of previously discarded activation records of the same process type. Activation records of active processes are linked together in a tree structure, rooted in the first activation record allocated for the kernel.

From time to time the model checker needs to visit each of the activation records. While it is possible to traverse the tree either recursively or iteratively, it is much easier to maintain an ordered list of pointers to the records. More conveniently, this list can also be used as a canonical representation of the state.

An action may: transform the local variables of the executing process, possibly change some variables in other activation records and affect the state of an channel. Instead of inspecting each activation record after an action has executed, the model checker records only the processes that are affected by an action, and only these processes are updated when inserting a new state.

The LF compiler identifies the processes which access variables outside of its own scope, and set the topmost bit of the pool number to indicate this to the model checker. Unfortunately

the current compiler does not attempt to predict exactly which processes may affect the value of a shared variable. Since the model checker does not have knowledge of which variables are accessed in an action, the model checker takes a conservative approach and updates all processes when a process which accesses shared variables executes an action.

This approach may be inefficient in many cases since more processes than are necessary are updated. The situation can be improved if more work is done in the compiler to identify the actions (rather than processes) which affect the value of shared variables.

The local state tables are organised as large closed hash tables, one for each process or channel type. This strategy was chosen to simplify the management of the local state tables. Since the compiler supplies the number of process types used in the program, the model checker can allocate the maximum table size (2^{16} entries) for each type during initialisation. This upper-bound on the table size may seem restrictive, but, we feel that any program of that size has a single component with more than 2^{16} entries, will more than likely be too large for our model checker to effectively handle.

Sharing a single table amongst all the instances of a process type does waste some slots. But this is an acceptable overhead since in some cases the same instance of a run-to-completion-process, occurs along different paths in the depth-first search and therefore will be stored only once in the local state table.

4.3.2 Global States

A global state describes the whole LF system at a single point in the execution. The global state is an ordered array of indexes into the local state tables, a signature of the state and a flag. The flag marks whether the state is available for replacement, or if the state is still on either the depth-first or component stack.

The signature describes the layout of the state, and is an ordered list of records which associate each index in the state with a process identifier and local state table. This information is crucial to uniquely identify two distinct states with the same index values. Instead of storing the signature explicitly in the state vector, the same technique that is used to store the local

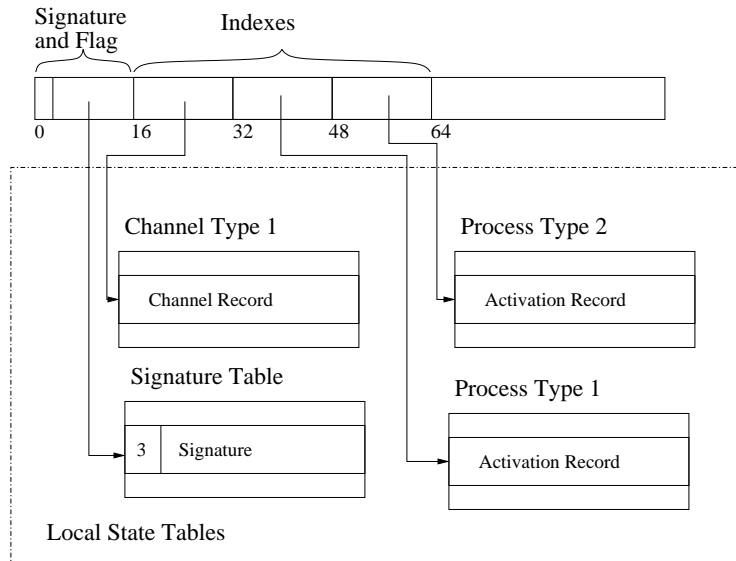


Figure 15: The structure of the state vector

states of processes are applied to the signature. Figure 15 show the structure of a state.

The indexes in the state vector are sorted in ascending order according to the process identifiers. Channel records are assigned negative channel identifiers, and this can be used to artificially partition the state vector into two parts, the indexes associated with channels and those of processes. When a process or channel is added or removed, the indexes are reorganised to ensure that the order is preserved in the state vector. This scheme is sufficient as LF programs generally allocate channel records during the initialisation of the modules; thereafter new channel records are infrequently added. New processes are always assigned higher process identifiers than those already in the state vector, and in most cases only the tail of the state vector grows and shrinks.

To avoid reassembling the state from scratch each time a new state is inserted, the model checker keeps a single copy of the most recent state either restored from the state cache, or inserted in the state cache. This allows only updating the indexes of the processes affected by the last action. This is an important optimisation since most LF actions only affect one or two processes and a channel.

To simplify the implementation of the state store, each global state entry has the same size.

As the number of processes in the system may vary during the execution, the signature of the state describes which of the indexes are currently in use. The maximum size of a state is a parameter that the user defines and must be large enough to accommodate the peak number of processes and channels, but small enough to minimise wasted indexes.

Restoring a state from the state cache is a simple process; using the entry in the signature state table, the indexes in the state vector is restored to the LF heap.

4.3.3 State Cache

Storing states encountered during on-the-fly generation of the state graph is essential to avoid redundant work. The main bottleneck in the LF model checker is the state management. The state store is organised as a large closed hash table and states are entered into the table as soon as they are generated.

To insert a state in the hash table, we compute two hash functions, h_1 and h_2 . The first value h_1 is used as the index for the state in the hash table. There are four cases to consider:

- If the indexed slot is empty, the state is inserted, and the index returned to the depth-first search engine.
- If the slot contains the same state, only the index is returned.
- If the slot contains another state, other potential slots are “probed”. The first slot probed is $h_1 + h_2$, then $h_1 + 4h_2$, then $h_1 + 9h_2$, and for the i^{th} probe, $h_1 + i^2h_2$, until a slot is found that is either empty or contains the same state.
- After a fixed number of probes K (which is at most the size of the hash table) we conclude that there is no space in the table.

When the number of states explored exceed the hash table size, the table is guaranteed to run out of slots. This situation can be handled in a number of ways:

- I End the search indicating that there is no more space for new states.

- II Ignore the state, and continue as if the state has been encountered before.
- III Continue generating states treating each state as a new state, but do not insert the state into the hash table.
- IV Replace an existing state in the hash table.

Continuing the depth-first search with option III will eventually be able to generate the entire state space, however, since no other states than those on the depth-first stack are stored, the amount of double work required will be prohibitively expensive. Furthermore, since our model checker does not store states directly on the depth-first stack this approach will not work.

Option III is definitely preferable to I, but results in a partial search of the state space. Ignoring potentially large parts of the state vector may result in false positives. Many model checkers rely on option IV, since overwriting an older, fully explored state, does not change the result of the verification. Option IV is generally known as state caching.

The number of probes needed to resolve a conflict can be used as a measure of the remaining available slots in cache. On average, longer probe sequences indicate that most slots in the table have been used.

Clearly, the value of K affects the performance of the cache. If K is too small states will be replaced while the cache is relatively empty. This leads to a large amount of redundant work. On the other hand, if K is too large, insertion will be relatively and unnecessarily slow once the cache is close to full. It is therefore important to find a suitable value for K ; usually this is done empirically. In our model checker the value of K is at most 10% of the cache size.

After K probes the first replaceable state in the probe sequence is replaced. A state is replaceable when it is neither on the depth-first stack nor on the component stack. In essence, any replaceable state in the probe sequence have the same probability to be revisited. Replacing the the earliest state along the probe sequence does however limit the length of the probe sequence if the new state will be visited again.

Chapter 5

Evaluation

During the design algorithms and data structures are selected based on conceptual merits. To judge whether the techniques perform and scale well, detailed measurements of the model checker performance is necessary. Such measurements of the components of the model checker can also be a great aid in identifying potential problems, and areas where the model checker can be improved. In this chapter we will gauge the performance of the LF model checker, specifically:

- establish that the model checker perform correctly,
- determine how much is gained by the coarse-grained actions in LF, and
- show that keeping blocked processes sorted in the channel queues reduce states.

Model checker performance is usually judged based on metrics such as the memory consumption and the throughput (in terms of states or transitions per second). Comparing two model checkers based solely on these metrics may be misleading because the internal workings of the two model checkers may differ significantly. In order to avoid this problem the LF model checker is measured against itself.

A model checker is a modestly complex software system, and testing a model checker is a crucial step in its development. A small error in any of the components of a model checker can invalidate the verification.

LF is an experimental system and the collection of examples is limited. Most of the programs written in LF has been formulated to illustrate a specific feature, and indeed LF has not yet been used to solve a large software problem. A selection of LF programs, which we feel covers most language features and range in size are used in the tests.

5.1 Establishing the Correctness of the Model Checker

The first task in evaluating the model checker is to gain confidence in the model checker implementation. This involves dissecting the components of the model checker and evaluating each in turn. However, the interaction of the components is usually where most errors are introduced. For the sake of brevity, we categorise the tests into correct state space generation and correct cycle detection.

State Space Generation

Correct state space generation is vital to the correctness of a model checker. The state generator consist of the LF program and runtime system, the state cache, and the depth-first search engine. The correctness of the runtime system and compiler falls outside the scope of this thesis, and we accept them as correct.

There are several approaches to determining the correctness of the state generator. Some of the techniques include the following:

- Using small models for which a state space can be manually constructed or inferred, and visually comparing the manual results to those generated by the model checker.
- Constructing models that will scale in a predictable manner. For example, if the model contains n identical independent processes with k actions each, the state space should contain k^n states.
- Replacing the underlying runtime system with graphs. Either random graphs, or state graphs from other tools. However, random graphs are not representative of state spaces [61].

The main concern in this model checker is the interaction between the runtime system and the depth-first search. The tests the state generator is subjected to, falls into the first two categories. The third is a more general testing technique that focuses on testing state space searching heuristics.

The model checker was modified to provide a state graph, which is compared to the manually derived state spaces. Most tests were constructed to include a significant number of system calls. The communication system calls, in particular were carefully examined, as these directly influence the set of enabled processes, and as such the structure of the state graph. The tests that were used can be divided into the following tests:

- process creation and termination, run-to-completion processes and concurrent processes;
- simple communication system calls;
- guarded communication system calls;
- all the system calls which do not explicitly affect the set of enabled processes; and
- tests which combine the above elements.

The model checker behaved as expected on all tests except for a handful of examples. Careful inspection of these examples showed that they all have the following code in common:

```
1 CHOICE
2   ch?msg;
3 OR
4   ch!msg;
5 END
```

This piece of code contains a non-deterministic choice between two communication statements. Instead of exploring all the successors of the communication statements first, the depth-first search explored non-deterministic choices first. This implies that instead of pairing each communication statement with all possible partners only the first partner was considered. The state generator was corrected to explore the transitions in the correct order, and the state generator subsequently passed all the tests.

```

1  IF (CycleDetection) THEN
2    IF (CStack[p].lowlink = p ) THEN  INC(SimpleSCCcounter);
3    sccsize := ctop-p;
4    WHILE (ctop >= p) DO
5      h := hash(CStack[ctop].state);
6      CHash[h] := CStack[ctop].next;
7      SC.Cached(CStack[ctop].state);
8      DEC(ctop);
9    END;
10   END;
11   IF (atop > 0) & (p = AStack[atop]) THEN
12     atop := atop -1;
13   END;
14   IF (TopState > 0) THEN Update(p); END;
15 END;

```

Figure 16: A snippet of the SCC detection algorithm

Cycle Detection Algorithm

Although the authors of [26] prove their algorithm correct, this does not mean that all implementations thereof are also correct. The model checker was extended to not only produce a state graph, but to also mark each state with the SCC to which it belongs. This state graph was then fed to a known, correct SCC detection algorithm and the results of the two implementations compared. A number of small tests constructed to scale in the number of SCCs was used for testing.

The algorithm seemed to perform as expected for the constructed models, and several of the larger existing models were used to test the algorithm further. In some instances the number of SCCs reported by the model checker did not match the number of SCCs reported by the reference implementation. A refinement of one of the examples led to the discovery of an error in the implementation of the algorithm. The structure of the initial test models caused the error to be missed in the first set of tests, since these only contained large SCCs with no cross edges, and only single successor loops.

Figure 16 shows an extract of the Pop operation responsible for detecting when an SCC has been fully explored. The error was caused by the swapping lines 10 and 14 around. This caused the algorithm to incorrectly conclude that an SCC with a state that has a cross edge,

that still has to be explored, is complete. After the bug was fixed, the algorithm behaved correctly.

The cycle detection algorithm was further tested by combining the algorithm with state caching. In some test cases, once states were replaced in the state cache, the component stack of the algorithm grew disproportionately, and unexpectedly large. After careful investigation the problem was identified: the state cache replaced states which were still on the component stack. The partial SCCs to which the replaced states belongs are never recognised as complete and therefore never removed from the component stack. The bug was fixed and the algorithm again performed as expected.

5.2 Performance

Determining the relative performance of a model checker is a difficult task as no two model checkers are exactly equivalent. Although it may seem acceptable to utilise some metric of throughput or memory utilisation to compare two model checkers this can be very misleading since the performance of a model checker rely on a combination of factors:

- Type of model checker: symbolic or explicit state. Symbolic model checking represents the state space extremely compactly and the verification algorithms differ significantly.
- State storage design: if the state store relies on implicit storage techniques, accessing and inserting a state may be costly, whereas with simple hashing reasonable bounds can be placed on the time required to insert and access a state.
- Whether utilising pre-compiled versus interpreted transition system, and the execution environment. Interpreted transition systems may incur a greater execution cost than transitions that execute directly on the underlying hardware.
- The cycle detection algorithm, NDFS versus SCC based. Although the cost of exploring transitions may be the same for both algorithms, the amount of work required to detect cycles varies significantly. In a NDFS based model checker in some cases the entire state

space may be visited twice, whereas SCC based model checkers require only a single pass.

Because we will measure our model checker against itself to determine how it performs, it is important to understand how each component of the model checker contributes to the execution profile. This allows us to discern which components affect the throughput and memory consumption.

5.2.1 Examples

The models used in the comparisons are variations of four basic examples: Peterson’s bakery algorithm (Peterson), the sliding window protocol (Sliding), an example of the dining philosophers problem (Philosophers) and producer/consumer system that communicate via a bounded buffer (Bounded Buffer). These examples roughly represent the combinations of the factors that influence the structure of LF programs:

- number of processes,
- the length of actions,
- the amount of variables per process, and
- whether the example uses message passing or shared variables.

In short the details of the examples can be summarised as follows: The columns of the

Model	Communication	Actions	Scaling
Peterson	Shared Variables	Long actions	Processes
Sliding	Message Passing	Short actions	Variables
Philosophers	Message Passing	Short actions	Processes
Bounded Buffer	Message Passing	Long Actions	Variables

Table 1: Some examples used in the Comparison

table represent, the model, the type of interprocess communication used, the length of the actions (short actions contain less than 20 machine instructions), and the factor in which the

example can scale. Each of the models were executed on an Intel Pentium 4 processor running at 2.8GHz, with an 800MHz front-side bus and 2gigabytes of DDR400 RAM. Timing of the LF model checker is provided by the Intel 8253 timer chip. Total time of the verification is measured from the point where the initial state is inserted in the state cache, up to when either a violation is reported, or the initial state is popped of the depth-first stack.

5.2.2 Profile

A good starting point in determining how the design decisions affect throughput, is to measure how each component of the model checker contributes to the runtime of the verification. A selection of programs which cover a significant range of LF language features and size were used to determine the contribution of the following:

- Pushing a state onto the depth-first stack and component stacks (Push).
- Compiling a state and inserting it in the state cache (Insert).
- Updating the local state tables (Local).
- Selecting processes to continue the depth-first search (SelectNext).
- Backtracking to an earlier state. Restoring the LF heap and removing the state from the depth-first stack. When cycle detection is enabled, several states may have to be removed from the component stack.
- Execution time of the actions. The time spent in the LF program or runtime system.

Figure 17 presents a summary of the tests. Each column of ticks represents one of the factors above. A tick mark in a column represents the relative contribution of the factor for a particular model. Although only a handful of examples are given, the behaviour tends to be similar for all models.

In most models, state operations — updates of local state tables and inserting new global states into the state cache — accounts for up to 90% of the total runtime. The time spent executing actions in most cases accounts for less than 10% of the total verification time.

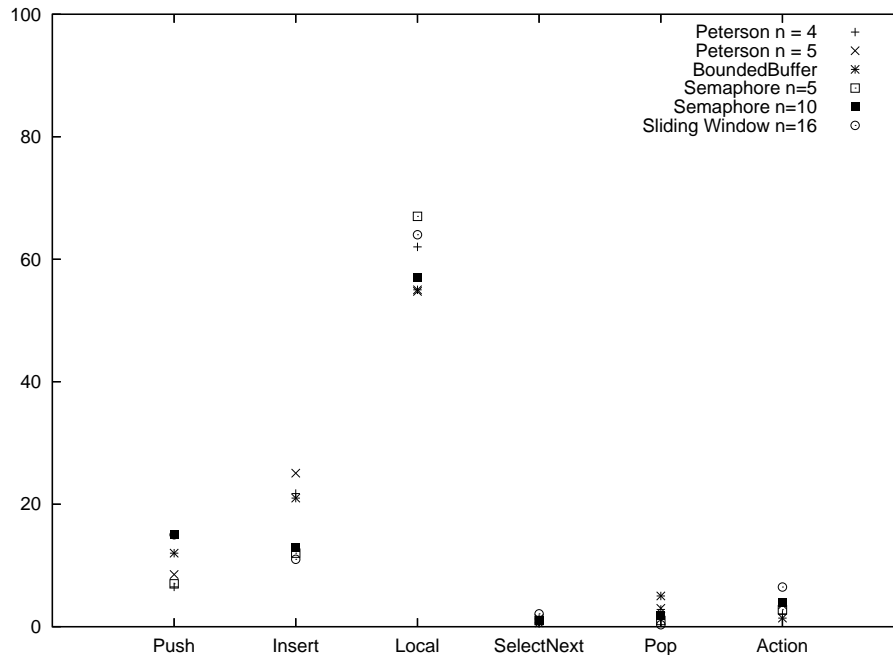


Figure 17: Profile of the system

Up to 75% of the time spent during verification is spent keeping the local state tables updated. Unfortunately the large activation records that LF programs require make this very slow. Reducing some of redundant information in the activation record could result in some gains in performance. One possible improvement is to remove the register scratch space from the activation records.

The scratch space is only used during the execution of an action. The LF compiler attempts to keep most of the variables used during an action in registers. Often while evaluating long expressions, the number of registers are insufficient and the scratch space is used to temporarily store the value of a register, making way for another variable. Although the scratch space comprises only roughly a third of the activation record header, eliminating it saves considerable space in the local state tables and improves the runtime considerably.

In contrast to state management the execution of actions contribute a very small amount to the total runtime of the verification. This is an good indicator that any increase in the length of an action that could reduce the number of states, would be justified.

5.2.3 Actions

The main state reduction technique in LF is the use of coarse-grained actions, which limit the interaction between processes. As we have seen in the previous section, the contribution of executing actions is negligible compared to state manipulation. To see how the technique performs and if it scales well we use a model of Peterson's algorithm. The implementation of this algorithm is perfect for this experiment as it relies on shared variables for communication rather than message passing which would add an extra layer of overhead. We know that the number of states in this example grows exponentially as more processes are added.

To determine the effect of large, coarse actions, we decided to break the actions down into much smaller units. This is achieved by inserting a Reschedule system call after every statement.

Table 2 shows the result of model checking four versions of the model. The models differ only in the number of processes, which is shown in the first column N . The last two columns show the metrics when the coarse- versus fine-grained actions. The metrics used in the comparison are the number of states and transitions, the maximum depth-first and component stack usage and the total verification time.

The number of states in the fine-grained solution grows rapidly and already for $N = 4$ the model checker needs to store more than 10^8 states and runs out of memory. In contrast to this, the coarse-grained actions show a much slower (although still exponential) growth rate, which allows the model checker to analyse the four-process example in a state store with at most 10^6 slots.

A fringe benefit of the coarse-grained actions, especially for this example, is that it reduces the mean size of SCCs in the example. In all cases the coarse-grained actions contained fewer and smaller non-trivial SCCs than the finer-grained examples. The effect of this is that the total stack usage increases as the granularity decreases. This is easily explained if we examine how the structure of the program has changed. The chains of single successor states in the fine-grained example is condensed in the coarser-grained solution leading to smaller SCCs.

N	Metric	Normal Actions	Fine grained Actions
2	States	85	313
	Transitions	169	625
	DFS Stack	32	108
	Component Stack	41	154
	Time(ms)	45	157
3	States	5594	87497
	Transitions	16780	262489
	DFS Stack	389	2866
	Component Stack	17404	137
	Time (ms)	210	7578
4	States	1094676	out of memory
	Transitions	2588315	-
	DFS Stack	8670	-
	Component stack	156087	-
	Time (ms)	40488	-

Table 2: Normal LF program vs. Fine grained

5.2.4 Sorted Channel Queues

As we have seen in Section 4.2.2, to avoid some of the redundant states caused by the ordering of the process activation records blocked and waiting in a channel queue, these entries are kept sorted. Keeping the entries sorted does incur a runtime penalty, but the question is whether this penalty is offset by a saving in states. The simplest approach to keeping the queues sorted is to use insertion sort based on the process identifier, which in the worst case would require comparing each new entry to all those already present in the queue.

To illustrate how much keeping the channel queues sorted influences the number of states, consider an example where multiple senders attempt to communicate with one receiver over the same channel. Although for this demonstration we have eliminated other factors, this is a typical scenario encountered in server/client architectures. It is a reasonable assumption that most larger LF programs will follow this design pattern, since process instantiation is relatively expensive [34] (it is cheaper to send a message to a server than to invoke a run-to-completion process to accomplish the same task), and no variables may be shared between modules. In the examples used in the comparison N represent the number of processes that attempt to transmit a signal over a shared channel and therefore also the maximum length

N	Metric	Sorted Queues	Unsorted Queues
4	Transitions	465	4385
	States	113	1221
	Verification Time (ms)	140	172
	Actions (ms)	16	19
5	Transitions	1563	61698
	States	307	14085
	Verification Time (ms)	174	781
	Actions (ms)	21	71
6	Transitions	5201	1054565
	States	857	203895
	Verification Time (ms)	199	8143
	Actions (ms)	22	675

Table 3: Savings incurred by sorting the queues

of the queue.

As the examples illustrate, sorting the queue entries reduces the number of states drastically as the maximum length of the queue increases, with a small increase in runtime per action.

If the first-in-first-out ordering of the processes waiting on a queue is preserved $\sum_{i=0}^N P_i^N$ states are required to store the different permutations of the processes. Sorting the queues maps each permutation to a canonical representation and in turn reduce the number of states drastically since now all possible combinations of processes can now be represented in $\sum_{i=0}^N \binom{N}{i}$ states. Obviously, this technique only produces a reduction in states when the maximum queue length is greater than two. In essence, sorting the channel queues is a form of symmetry reduction. In some cases this technique may even increase the reduction that is possible when applying partial orders.

5.3 Comparison to Other Model Checkers

Although comparing two model checkers directly does not always give a credible view of the relative performance of the two model checkers, we give the result of some examples of LF programs that have nearly equivalent Promela specifications. In essence, the comparison is unfair since the LF programs are much more coarse-grained than the Promela models, and often it is not possible to transform the Promela statements to follow the same structure as

the LF actions.

The following model were used to compare LF and SPIN; the results are shown in Table 5.3:

- Peterson: Peterson’s algorithm for four processes. The LF model was translated from the Promela specification provided with the SPIN tool. The actions in LF were carefully constructed to only reschedule between statements which access the shared variables.
- BoundedBuffer: A simple producer consumer model that communicates via channels and a buffer process, with 20 slots.
- DiningPhil : An example of the dining philosophers. Each of the five philosophers and forks are modelled as processes. Critically, this example has no local variables and relies solely on message passing.
- Semaphore : A model of a semaphore, with five processes competing to access the semaphore. The semaphore is modelled by a process and accessed via a channel.
- Eratosthenes : The sieve of Eratosthenes translated from the Promela model.
- SWP : An sliding window protocol implementation.

The first column in Table 5.3 indicates the which model is used in the comparison, the second column lists the metrics used in the comparison, the third column (LFMC) show the results of the LF model checker. The fourth and fifth columns (SPIN and SPIN+PO) respectively show the results for the SPIN tool, with and without partial order reduction enabled.

The metrics used in this comparison, provide the basic information of the model, the number of states and transitions, the total memory usage required to perform the verification, the compressed state vector size (and in brackets the uncompressed state vector size), the maximum depth reached and the total time of the verification. The figures for SPIN and SPIN+PO are exactly as reported by the tool itself. The total memory usage for the LF model checker is the combined total of the memory used for: local states, global states, and the maximum stack usage of the depth-first and component stacks. Each verification run of an LF model, is perfect, in that no state is replaced in the state cache.

Model	Metric	LFMC	SPIN	SPIN+PO
Peterson	States	1094676	6.5×10^6	533083
	Transitions	4378701	24×10^6	888657
	Memory Total	21×10^6	279×10^6	28×10^6
	State vector Size	16 (702)	32 (32)	32 (37)
	Maximum Depth	8670	1297629	165342
	Time (s)	40.5	98.02	1.15
BoundedBuffer	States	2571	8709	5987
	Transitions	4428	16269	9303
	Memory Total	376896	1.53×10^6	1.12×10^6
	State vector Size	20 (732)	156 (392)	169 (392) Bytes
	Maximum Depth	1303	3289	2617
	Time (s)	0.25	0.11	0.09
DiningPhil	States	437127	321	175
	Transitions	2802301	1057	511
	Memory Total	20×10^6	102000	102000
	State vector Size	40 (1724)	76 (288)	76 (505)
	Maximum Depth	78572	307	197
	Time (s)	24.8	0.01	0.01
Eratosthenes	States	7758	25295	2093
	Transitions	29431	90558	2571
	Memory Total	380902	2.998×10^6	307000
	State vector Size	44 (388560)	117 (280)	134 (280)
	Maximum Depth	86	288	288
	Time (s)	0.98	0.32	0.03
SWP	States	122385	159896	72268
	Transitions	439306	597975	153779
	Memory Total	4×10^6	20.4×10^6	10.7×10^6
	State Vector Size	32 (1428)	135 (156)	135 (156)
	Maximum Depth	2206	1592	821
	Time (s)	3.1	2.27	0.54

Table 4: Statistics of several models

Leaving the DiningPhil model to one side for now, LF outperforms SPIN without partial order reduction in every one of the remaining models. The latter generates more states, by a factor that ranges from 1.3 to 5.9. (The same is true for the transitions). However, if we examine the transitions explored per second ratio, we see that SPIN beats LF in all cases. The SPIN tool manages to explore between 1.5×10^5 and 2×10^6 transitions, whereas LF only manages to explore between 2×10^4 and 10^6 transitions per second. This is largely attributable to the size of the uncompressed state vectors in the LF model checker that are at least twice as large as in SPIN and in some cases up to a factor 20 larger.

When partial order reduction in SPIN is used, the picture changes somewhat. LF still generate fewer states in the BoundedBuffer example (factor 2.3 fewer), but in all other cases this is not true. Even with partial orders, SPIN still require significantly more storage space. One must bear in mind that there is no reason why partial order reduction will not work as well in LF as in in SPIN. The Eratosthenes example is a perfect case for partial orders in LF. Since almost all the transitions are independent of each other, SPIN was able to reduce the state space by 90%.

In most cases the number of states of the LF models are less than those of the equivalent Promela models except the DiningPhil model. This example was chosen to illustrate the worst-case behaviour of LF. Since the example relies solely on interprocess communication, any savings to be had by the longer actions in LF is removed. Instead the model illustrates how much overhead is incurred by storing the information in the channel records. Secondly, the semantics of LF and Promela differ in their definition of synchronous message passing, the SPIN verifier will only execute an communication statement if and only if an complementary statement is enabled in the same state, in which case they both execute simultaneously. Whereas in LF the a communication statement may execute regardless of whether an complementary action is ready to execute, and if there is no avialable partner, the process blocks.

Chapter 6

Conclusion

This thesis examined model checking as a tool to derive verified implementations. The approach employed in the LF system is that the design was carefully considered to construct a safe programming language which lends itself well to model checking. The LF model checker is designed to serve as a replacement of the normal runtime environment, and provides the same support as the existing framework, while extending it with state space exploration and on-the-fly model checking. This allows the model checker to detect violations of temporal logic properties in the actual implementation. In cases where the implementation yields a closed system, direct model checking of the program is possible. However, when an open system is considered a representation of the external environment is necessary to close the program.

From the discussions in Chapters 2 through 5 we can sketch the following picture of the LF model checker:

- Instead of relying on an abstract machine to execute the model, the computer hardware and the system calls of the normal runtime system provide the execution environment. LF programs are executed in the usual manner when model checking, except that all possible scheduling possibilities are evaluated at the end of each action.
- The idea of structuring the program code into several coarse-grained actions has several

benefits. For embedded applications the fine control over the length of actions enable developers to structure code to achieve predictable schedules and behaviours. More importantly, without the coarse-grained actions model checking LF programs would be prohibitively expensive, and even small programs would be nearly impossible to handle due to the number of possible interleavings of machine instructions. In some cases the coarse actions in LF fare even better than partial order reduction techniques in SPIN.

- Purpose-built model checkers such as the LF model checker have several benefits. From a model checker design and heuristic point of view, the knowledge about the structure of programs that are executed on the system can be used to improve or optimise the model checker. A good example of this is the sorting the channel records, in order to collapse equivalent states.
- Designing and implementing a model checker to form part of an existing system presents some unique challenges. One example is defining which features of the original kernel should be kept intact and which the model checker should abstract. In the LF kernel, only the scheduling and synchronisation policies were abstracted. Other problems range from simple implementation details to architectural differences. A model checker requires a compact representation of the state of the system — with little to no redundant information — while often runtime environments require some redundancy for efficiency gains. One clear example in the LF system is the channel records, where some model checker efficiency was sacrificed for runtime efficiency.
- State storage: Model checking is resource-intensive, and state storage requires a lot of memory. In order to represent each state as compactly as possible the collapse compression technique is used. However, providing program model checkers on very limited hardware may require techniques such as hash compaction. In reality, most programs will exhibit so many states, that inevitably the LF model checker will have to resort to bitstate techniques to analyse large programs.
- The improved version of Tarjan’s algorithm is able to detect cycles much earlier than the original implementation. Although a nested depth-first search was not implemented,

the fact that a violation can be reported (by the improved algorithm) as soon as each of the states that form the violation is explored, would require less stack space.

6.1 Related and Future Work

In order to facilitate development of reliable software, aside from the model checker, a remote debugger was designed and implemented for the LF system. Earlier projects included IP protocol stack for the LF system and a Network Traffic Protocol analyser. One project that can be of great value when combined with the model checker is the LF Slicer [51].

Program Slicing

The complexity contained in real software systems are due to the multiple tasks even simple systems have to perform. Slicing is a technique used to extract a smaller program by identifying and removing statements and variables that do not affect the values of a set of other variables at a specified point in the program. Slicing was originally conceived as a debugging tool to extract a smaller portions of code to find the cause of an error, and has become a valuable tool in verification. The LF slicer [51], is a static slicer for concurrent programs which can be used to reduce the complexity of LF programs considerably.

To illustrate how the application of this technique can benefit software model checking, consider the classic example used in slicing literature, the product/sum example. The example has two processes, the first which computes two values, $\sum_{j=1}^n j$ and $n!$ which are then transmitted to a client process over two different channels. Applying the slicing algorithm on the result of the factorial only removes all the calculation and the message passing statements to transmit $\sum_{j=1}^n j$. As we can see in Table 5 even for this small example, where $n = 10$, the savings are significant, eliminating up to half the states. Slicing is a valuable tool when model checking programs, since only the the behaviours and variables most relevant to the property under investigation can be sliced out of a large implementation.

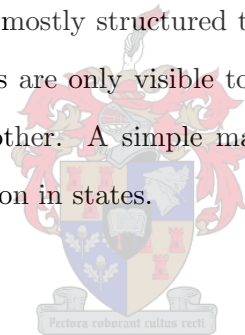
Example	States	Transitions	Maximum Depth	SCCs
Original	33	55	31	23
Sliced	19	30	9	17

Table 5: Reachable states of a small sliced example

Partial Order Reduction Techniques

Partial order reduction techniques [28, 31, 62, 63] reduce the number of states and transitions that a model checker has to explore. Broadly speaking, partial order techniques exploit the fact that a single state in a state graph may be reached through several equivalent paths. Two paths are equivalent if the order in which the transitions are explored to reach a state do not matter.

Adding partial order reduction to the LF model checker will significantly improve the LF model checker. As LF programs are mostly structured to interact through message passing, the bulk of variables in LF programs are only visible to a single process, and as such most actions do not interfere with each other. A simple manual exercise using for a small LF example showed up to a 70% reduction in states.



6.2 Remarks

A model checker for the LF language was successfully implemented. The implementation is structured in a modular to allow easy extension of the model checker to integrate new techniques. Although other model checkers also include tools to enable grouping adjoining statements into atomic actions, the aggressive approach taken in the LF system lies at the heart of the success of this model checker.

Bibliography

- [1] H. Barringer and W. Visser. Memory efficient state storage in SPIN. In *Proceedings 2nd Workshop on the SPIN Verification System*, volume 32, pages 185–203. American Mathematical Society, May 1997. ISBN 0820180680. (32, 34, 48)
- [2] Gerd Behrmann, Kim Guldstrand Larsen, and Radek Pelánek. To store or not to store. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 433–445. Springer, 2003. (36)
- [3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions of Computers*, C-35(8):677–691, 1986. (9)
- [4] D.B. Bull. A comparison of two different model checking techniques. Master’s thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 2002. (15, 25)
- [5] 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. (9, 10)
- [6] S. Christensen, L.M. Kristensen, and T. Mailund. A sweepline method for state space exploration. In *TACAS’2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464, 2001. (37)

- [7] E. M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001. (10)
- [8] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. (13)
- [9] C. Colby, P. Godefroid, and L.J. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998. (46)
- [10] J. C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineerings*, pages 439–448, June 2000. (26)
- [11] T.H. Cormen, C.E. Leieron, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachutes, 1990. (41)
- [12] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. In *Proceedings of the 2nd International Conference in Computer Aided Verification (CAV'02)*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, June 1990. (12, 39, 42)
- [13] J. M. Couvreur. On-the-fly verification of temporal logic. In *Proceedings of FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, September 1999. (43)
- [14] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997. (13)
- [15] E. W. Dijkstra. *Finding maximum strong components in a directed graph (EWD376)*, chapter 24. Prentice-Hall, 1976. (43)
- [16] D. L. Dill and U. Stern. Combining state space caching and hash compaction. In Bernd Straube and Jens Schoenherr, editors, *4. GI/ITG/GME Workshop zur Methoden des*

- Entwurfs und der Verifikation Digitaler Systeme*, pages 81–90, Kreischa, 1996. Shaker Verlag, Aachen. (26)
- [17] P. Dillinger and P. Manolios. Fast *and* accurate bitstate verification for SPIN. In *11th SPIN Workshop on Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2004. (37)
- [18] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Logic in Programs: Workshop, Yorktown Heights, NY*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, December 1981. (7, 9)
- [19] 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. (7)
- [20] E.A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, Volume B: Formal Methods and Semantics:995–1072, 1990. (8)
- [21] E.A. Emerson and A. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996. (13)
- [22] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *Proceedings of the 11th International SPIN workshop on Model Checking Software*, pages 154–167. Springer-Verlag, April 2004. (40, 41)
- [23] J. Geldenhuys. State caching reconsidered. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 23–39, April 2004. (36)
- [24] J. Geldenhuys and P.J.A. de Villiers. Runtime efficient state compaction in SPIN. In *Proceedings: 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 12–21, 1999. (31)
- [25] J. Geldenhuys and A. Valmari. Tarjan’s algorithm makes on-the-fly ltl verification more efficient. In *Proceedings of the 10th International Conference on Tools and Algorithms*

- for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer-Verlag, March/April 2004. (42, 43)
- [26] J. Geldenhuys and A. Valmari. More efficient on-the-fly ltl verification with tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, November 2005. (42, 48, 56, 69)
- [27] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *15th Workshop on Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall. (10)
- [28] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem*. PhD thesis, University of Liège, December 1994. (13, 83)
- [29] P. Godefroid and G. Holzmann. On the verification of temporal properties. In *Proceedings of IFIP Symposium on Protocol Specification, Testing and Verification.*, pages 109 – 124, Liege, Belgium, June 1993. (42)
- [30] P. Godefroid, G. Holzmann, and D. Pirottin. State Space Caching Revisited. In *Proceedings: Computer Aided Verification*, pages 175–186, July 1992. (35)
- [31] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *6-th IEEE Symposium on Logic in Computer Science*, pages 406–414, Amsterdam, 15-18 July 1991. (83)
- [32] Patrice Godefroid. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design*, 26(2):77–101, 2005. (26)
- [33] J-C. Grégoire. State Space Compression in SPIN with GETS. In *Proceedings of the Second SPIN workshop*, Rutgers University, New Brunswick, New Jersey, August 1996. (34)
- [34] L. Grobler. A kernel to support computer-aided verification of embedded software. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, April 2006. (2, 23, 75)

- [35] P. Brinch Hansen. Efficient Parallel Recursion. *ACM SIGNPLAN Notices*, 30(12):9–16, December 1995. (23)
- [36] K. Havelund and W. Visser. Program model checking as a new trend. In *Software Tools for Technology Transfer: 7th International SPIN workshop.*, volume 4(1), pages 8–20, Stanford,CA, 2002. (23)
- [37] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, 1969. (7)
- [38] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985-2004. (6)
- [39] G. Holzmann. *Trends in Software Verification*, volume Lecture Notes in Computer Science No. 2805, pages 40–50. Springer-Verlag, 2003. (23, 26)
- [40] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The Spin Verification system: Proceedings of the second SPIN workshop*, pages 23–32, 1996. (40, 41)
- [41] G. J. Holzmann. Automated protocol validation in *argos*. *IEEE Transactions on Software Engineering*, 13(6):683–696, 1987. (35)
- [42] G. J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. *Software Tools for Technology Transfer*, 2(3):270–278, November 1999. (34)
- [43] G.J. Holzmann. An Improved Reachability Analysis Technique. *Software Practice and Experience*, 18(2):137–161, February 1988. (37)
- [44] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991. (37)
- [45] G.J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings: SPIN97 Workshop*, Enschede, The Netherlands, April 1997. University of Twente. (31, 32)
- [46] G.J. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. (25, 29, 32)

- [47] G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, April/June 2000. (26)
- [48] O. Kupferman and M.Y. Vardi. Relating linear and branching model checking. In *IFIP Working Conference on Programming Concepts and Methods*, pages 304–326, New York, June 1998. (8)
- [49] Lamport. “sometime” is sometimes “not never”: On the temporal logic of programs. In *POPL ’80: 7th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 174–185, 1980. (7, 9)
- [50] D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie-Mellon University, 1993. (13)
- [51] F. Louw. A slicer for LF. Master’s thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 2006. (46, 82)
- [52] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison-Wesley, 1992. (14)
- [53] Z. Manna and A. Pnueli. Verification of concurrent programs, part I: the temporal framework. technical report STAN-CS-81-836. Technical report, Department of Computer Science, Standord University, July 1981. (7)
- [54] Zohar Manna and Amir Pnueli. *The temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, New York, 1991. (8)
- [55] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993. (10)
- [56] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI ’02: Proceedings of the 5th Symposium on Operating Systems design and Imlementation*, volume 36, pages 75–88, 2002. (26)
- [57] N. Wirth and J. Gutknecht. *Project Oberon*. Addison-Wesley, 1992. (14, 49)

- [58] S. Owiki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–240, 1976. (7)
- [59] P. B. Hansen. Joyce a programming language for distributed systems. *Software Practice and Experience*, 17(1):29–50, 1987. (15)
- [60] B. Parreaux. Difference compression in SPIN. In *4th SPIN Workshop*, 1998. (34)
- [61] R. Pelánek. Typical structural properties of state spaces. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 5–22. Springer, 2004. (67)
- [62] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996. (13, 83)
- [63] Doron Peled. All from one, one for all: on model checking using representatives. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag. (13, 83)
- [64] A. Pnueli. Temporal logics of programs. *Proceedings 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977. (7)
- [65] S. Löffler and A. Serhrouchni. Creating Implementations from Promela Models. In *Proceedings 2nd Workshop on the SPIN Verification System*, volume 32, pages 72–80, Rutgers University, New Jersey, USA, May 1997. American Mathematical Society. (25)
- [66] E. Sami and P. Jean-François. Memory efficient state space storage in explicit software model checking. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, pages 43–57. Springer-Verlag, 2005. (34)
- [67] S. Schwoon and J. Esparza. *A Note on On-The-Fly Verification Algorithms*. Technical Report Technical Report No. 2004/06, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, University of Stuttgart,

- Institute of Formal Methods in Computer Science, Software Reliability and Security, November 2004. (41)
- [68] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computer and Mathematics with Applications*, 7(1):67–72, 1981. (41)
- [69] A.P. Sistla and P. Godefroid. Symmetry and reduced symmertry in model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(4):702–734, 2004. (13)
- [70] R. Swart. A Language to Support Verification of Embedded Software. Master’s thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 2003. (2, 20)
- [71] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972. (41, 42)
- [72] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Desing*, 1(1):297–322, 1992. (13)
- [73] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998. (13)
- [74] F. van Riet. LF: A Language for Reliable Embedded Systems. Master’s thesis, University of Stellenbosch, Stellenbosch 7600, South Africa, December 2001. (2)
- [75] M. Y. Vardi, P. Wolper, and A.P. Sistla. Reasoning about infinite computations. In *Proceedings of the 24th IEEE Symposium on the Foundations of Computer Science*, pages 185–194. IEEE Computer Society Press, 1983. (10)
- [76] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of a Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986. (10, 11, 38)

- [77] W. Visser, K. Haveland, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003. (26)
- [78] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker. In *Proceedings of the Workshop on Advances in Verification*, Chigaco, Illinois, July 2000. (26)
- [79] P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proceedings Computer Aided Verification*, volume 693 of *Lecture Notes in Computer Science*, pages 59–70, Elounda, Crete, June 1993. Springer-Verlag. (37)

