

# Model Checking for a Functional Hardware Description Language

by

John N. Foster  
Emmanuel College

A Dissertation  
Submitted in partial fulfillment of  
of the requirements for the Degree of Bachelor of Arts  
in Computer Science

Computer Laboratory  
CAMBRIDGE UNIVERSITY  
Cambridge, UK  
May 16, 2002



## Abstract

We present a verification tool for the functional hardware description language SAFL. Model checking [JGP99] is a popular verification technique. Given a model of a finite state transition system, and a temporal logic specification, a model checker determines if the model satisfies the specification. Advantages of model checking over other verification techniques are that it is fully automatic, and can produce counter examples for false specifications.

We have developed a model checker which uses the SAFL language as the modelling language. SAFL [MS00] is a functional hardware description language. As a result of this project, the same SAFL description can be used to describe a circuit and as a model for model checking. No separate model of the system is needed. It is hoped that this will make verification a regular part of the development process for SAFL programs. Just as a static type checker enforces safety properties for a programming language, we would like to see our model checker used to verify user-defined temporal properties before compiling SAFL programs to silicon. While limitations in model checking technology, (*i.e.*, the state explosion problem), make achieving this goal unlikely for larger programs, techniques such as abstraction and a translation to an industrial-grade symbolic model checker, SMV, can be used to overcome them in many cases.

## **Acknowledgements**

The author wishes to thank Professor Mike Gordon for initiating, overseeing, and supervising this project as well as for helpful comments on a draft of this dissertation. Thanks is also due to Dr. Alan Mycroft for ideas and consultation during the execution of the project. Richard Sharp provided his SAFL compiler and many helpful hints about its operation. Finally, thanks to the late Dr. Herchel Smith for generous financial support.

# Proforma

Name:	J. N. Foster
College:	Emmanuel College
Project Title:	Model Checking for a Functional Hardware Description Language
Examination:	Computer Science Tripos Dissertation
Word Count:	11,763 ( <code>detex 0*.tex   wc -w</code> )
Project Originator:	Dr M. J. C. Gordon
Supervisor:	Dr M. J. C. Gordon

## Original Aims of the Project

The original goal of this project was to produce a tool for verifying properties of SAFL programs. More specifically, we aimed to build a model checker which uses SAFL, a functional hardware description language, as the modelling language.

## Work Completed

We present a complete explicit state model checker for SAFL. First, we use a modified version of Sharp's SAFL front-end to produce parse trees annotated with temporal logic specifications. We then compile this language to an intermediate language, SAFLASM. From this intermediate representation, we infer the behaviour of the system, producing a finite-state transition system. Finally, a model checker verifies the temporal logic specification, producing an execution path as a counter-example if the property is not true of the system.

Additionally, we have implemented a translator from SAFLASM to the input language of SMV, an industrial grade symbolic model checker. This allows the space limitations typically associated with explicit state model checking to be avoided.

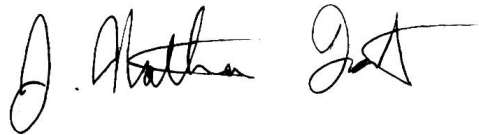
## Special Difficulties

No special difficulties were encountered in the execution of this project.

## Declaration

I, J. N. Foster of Emmanuel College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

A handwritten signature in black ink, appearing to read 'J. N. Foster', followed by a stylized flourish or second signature.

Date May 16, 2002

The SAFL front-end is reused from Richard Sharp's SAFL compiler. Also, the algorithms used for compiling SAFL to SAFLASM and for building the model checker are adapted from existing algorithms. These are cited appropriately in the dissertation.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Model Checking . . . . .	2
1.2	SAFL . . . . .	7
1.3	Project Goals . . . . .	8
1.4	Dissertation Organisation . . . . .	8
<b>2</b>	<b>Preparation</b>	<b>9</b>
2.1	Requirements . . . . .	9
2.1.1	Core Behaviour . . . . .	9
2.1.2	Code Reuse . . . . .	10
2.1.3	Practical Expressibility . . . . .	10
2.1.4	Efficiency . . . . .	10
2.1.5	Extensions . . . . .	10
2.2	Further Design . . . . .	11
2.2.1	Transition Systems . . . . .	11
2.2.2	Choosing a Semantics . . . . .	12
2.2.3	Extracting Kripke Structures from SAFL Programs . . . . .	13
2.3	Programming Practices . . . . .	14
<b>3</b>	<b>Implementation</b>	<b>15</b>
3.1	Front End . . . . .	15
3.1.1	Translate Equivalent Forms . . . . .	17
3.2	Translation to Intermediate Language . . . . .	18
3.2.1	Optimisation . . . . .	22
3.3	Building the Kripke Structure . . . . .	22
3.3.1	Non-Determinism and Branching . . . . .	23
3.4	Verification . . . . .	24
3.5	Producing Counter Examples . . . . .	26
3.6	Translation to SMV . . . . .	27
3.6.1	SMV Syntax . . . . .	27
3.7	Modules . . . . .	28

<b>4</b>	<b>Evaluation</b>	<b>31</b>
4.1	State Explosion . . . . .	31
4.2	Verifying Programs . . . . .	33
4.2.1	Simple Addition . . . . .	33
4.2.2	Counter . . . . .	34
4.3	Extensions . . . . .	36
4.4	Verification by translation to SMV . . . . .	37
4.5	Counter Examples . . . . .	37
<b>5</b>	<b>Conclusions</b>	<b>39</b>
5.1	Future Work . . . . .	40
5.2	Summary . . . . .	41
<b>A</b>	<b>SAFL</b>	<b>45</b>
<b>B</b>	<b>SAFLasm</b>	<b>49</b>
B.1	Syntax . . . . .	49
B.2	Operational Semantics . . . . .	50
B.3	Translation of SAFL to SAFLASM . . . . .	51
<b>C</b>	<b>Model Checking Algorithm</b>	<b>53</b>
<b>D</b>	<b>Example: SMV Translation</b>	<b>55</b>
<b>E</b>	<b>Project Proposal</b>	<b>61</b>



# List of Figures

1.1	Graphical Representation of a Kripke Structure . . . . .	3
1.2	A system satisfying $\text{AF } p$ . . . . .	4
1.3	Partial unwinding of Kripke structure . . . . .	5
2.1	SAFL Compiler Stages . . . . .	12
2.2	Design diagram for extracting Kripke structure from SAFL descriptions . . . . .	13
3.1	Typical Kripke structure for a SAFLASM program . . . . .	25
3.2	Final design for SAFL Model Checker . . . . .	29
4.1	Data for n-bit multiplier Kripke structure . . . . .	32
4.2	Graph for n-bit multiplier Kripke structure . . . . .	33



# Chapter 1

## Introduction

*Ordinary language is totally unsuited for expressing what physics really asserts, since the words of everyday life are not sufficiently abstract. Only mathematics and mathematical logic can say as little as the physicist means to say.*

– Bertrand Russell

---

The correct design and implementation of digital systems is a hard task. As systems become more complicated, it becomes increasingly difficult to engineer them to have their desired behaviours. Software engineering offers techniques for coping with these challenges. However, they do not solve the problem completely; informal declarative specifications are unable to express the intricacies of complex systems precisely. As Russell puts it, “the words of everyday life are not sufficiently abstract”. Formal logic is needed to establish safety properties for critical applications. While an error in a airline ticketing system may be undesirable, its ill effects are correctable. A similar error in an air traffic control system is unacceptable. As digital systems are now prevalent in many parts of everyday life (*e.g.*, in cars, microwaves, mobile phones), it is imperative that we have confidence they are safe. Thus, verification techniques based on formal logic are an increasingly necessary part of the systems development process.

Formal techniques provide a rigid and disciplined way to reason about systems. The techniques that are together called “formal methods” vary greatly. At a basic level, computer scientists often provide pencil and paper proofs of important properties of systems. For example, program language designers usually give a proof of type soundness that states that certain kinds of errors will not occur in the execution of well-typed programs. These proofs are very useful, and have been used to establish important safety properties for many systems. However, they rely on an expert to model the important

characteristics of the system and to produce the correctness proof (usually entirely by hand!). These proofs are both costly and error-prone, as they are generated by humans. As a result, computer scientists have developed verification systems that can produce (at least partial) correctness proofs automatically.

There are three key advantages to automatic verification. First, because the deductions are performed mechanically, there is less room for error than in hand-produced proofs. Second, many of the tedious cases of a proof can be handled automatically. Thus, the expensive expert is only needed for a reduced set of cases. Third, automatic techniques can sometimes be used to produce proofs directly from a specification of the system itself. This makes it possible to perform the verification on a description of the system itself, rather than on a simplified model. Or at the very least, the abstraction needed to produce the model can be automated, and performed in a principled rather than ad hoc fashion. Indeed, a key result of this project is in producing a verification mechanism where the description language for the system *is* the modelling language.

In this project, we explore a particular automatic verification technique known as model checking. This dissertation describes our work building a tool to verify temporal logic specifications of hardware circuits described in a functional hardware description language, SAFL.

## 1.1 Model Checking

Model checking is an automatic technique for proving properties about finite state systems. Given a model of a system, and a specification, a model checker automatically determines if the system satisfies the specification. Thus, in order to verify the behaviour of a system, it is only necessary to produce a model of the system  $M$ , and to express the desired property  $p$  in a suitable specification logic; the model checker does the rest.

### Modelling

Most model checkers work on a kind of model known as a Kripke structure, a description of a finite-state transition system. Clarke et. al [JGP99] give a formal definition as follows. A Kripke structure is a quadruple  $M = (S, S_0, R, L)$  and a set of atomic propositions  $AP$  (simple facts that are true of states in  $S$ ):

$S$	a set of states
$S_0 \subseteq S$	a set of initial states
$R \subseteq S \times S$	the transition relation
$L : S \mapsto \mathcal{P}(AP)$	maps states to sets of atomic propositions

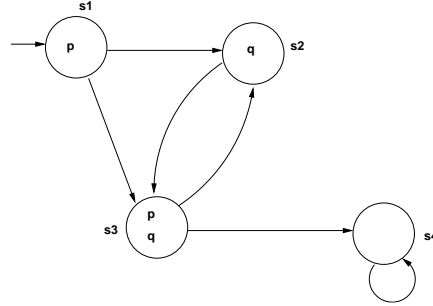


Figure 1.1: Graphical Representation of a Kripke Structure

Kripke structures provide a general way for expressing the transitional behaviour of systems. Consider the following Kripke structure over the set of atomic propositions  $\{p, q\}$ :

$$\begin{aligned}
 S &= \{s_1, s_2, s_3, s_4\} \\
 S_0 &= \{s_1\} \\
 R &= \{(s_1, s_2), (s_1, s_3), (s_2, s_3), (s_3, s_2), (s_3, s_4), (s_4, s_4)\} \\
 L &= L(s_1) = \{p\} \quad L(s_2) = \{q\} \quad L(s_3) = \{p, q\} \quad L(s_4) = \{\}
 \end{aligned}$$

As well as representing the atomic propositions that are true in each state, this Kripke structure captures the transition behaviour of the system. Thus, it is well suited to providing a model for reasoning about temporal properties of systems.

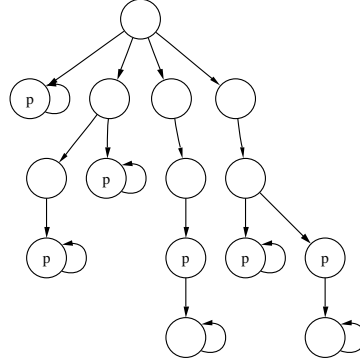
Kripke structures are often represented graphically; a sketch of the system above is shown in Figure 1.1. Each state is a node in the directed graph and is labelled with the atomic propositions true in it. Transitions in  $R$  are shown as edges between nodes. The start states in  $S_0$  are shown as nodes with an incoming edge.

These structures are very general, and can be used to represent a wide variety of systems. However, it is not always obvious which features of the actual system should be included in the formal model. The relevance of the verification results only apply to the formal model. Thus, deciding the features to include or abstract away is an important decision in the verification process.

## Specification

Specification logics give a syntax and semantics for expressing properties about systems. In the traditional approach to specification for programs, Floyd-Hoare logic [Hoa69], programs are annotated with predicates that are then proved using some axioms and rules. So-called Hoare triples have the general form

$$\{P\} \text{ cmd } \{Q\}$$

Figure 1.2: A system satisfying  $AF\ p$ 

and informal semantics that if  $P$  holds and the execution of  $cmd$  terminates, then  $Q$  holds after the execution. Hoare logic is generally useful for reasoning about output values from programs. However, in many applications, particularly in the hardware domain, temporal properties are just as important. For example, one might want to ensure that a value is *never* zero during a computation. Using Hoare-style logics, it would be difficult to formulate a specification of this property.

Temporal logics provide operators that make specifying properties which involve time components easy. A logic that is often used in model checking is Computation Tree Logic (CTL). Familiar first order logics, have connectives ( $\&$ ,  $\vee$ ,  $=$ ,  $\Rightarrow$ , ...) and quantifiers ( $\forall$ ,  $\exists$ ) for expressing symbolically the truth values of propositions or predicates. CTL includes additional forms such as  $AF\ \phi$ , which is true of states  $s$  where  $\phi$  is true in some future state along all possible computation paths starting with  $s$  (a system whose root state satisfies  $AF\ p$  is shown in Figure 1.2)

CTL properties are generally interpreted as applying to infinite computation trees. We can obtain an infinite tree by unwinding a Kripke structure, following each possible transition in  $R$  beginning with the start states. A partial unwinding for the Kripke structure shown in Figure 1.1 is given in Figure 1.3.

The syntax of CTL forms  $\phi$  is given by the following grammar:

$$\begin{aligned} \phi ::= & \top \mid \perp \mid p \mid EX\ \phi \mid AX\ \phi \mid EF\ \phi \mid AF\ \phi \mid EG\ \phi \mid AG\ \phi \\ & \mid E\ \phi\ U\ \phi \mid A\ \phi\ U\ \phi \mid \phi\ \&\ \phi \mid \phi\ \vee\ \phi \mid \neg\phi \end{aligned}$$

Informally,  $A$  is analogous to a  $\forall$  quantifier for path properties. Similarly,  $E$  is analogous to  $\exists$  for path properties. The modality  $X$  is true of states where a properties that hold in a “next” state,  $F$  for states where the property is true in some “future” state, and  $G$  for states with a properties that is true “globally”. Finally,  $U$  is true of states where one property is true “until” another property holds.

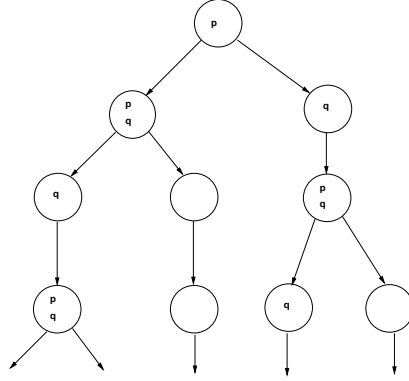


Figure 1.3: Partial unwinding of Kripke structure

CTL can be used to express many useful properties of finite state transition systems. The following CTL formulas are all true of the system,  $M$ , modelled in Figure 1.1 (we write  $M, s \models \phi$  if  $\phi$  is true of  $s$ ):

$M, s_1 \models \text{AF } \neg p$ : On all paths from  $s_1$ , there is a future state where  $p$  is not true.

$M, s_1 \models \text{EX } p \ \& \ q$ : From state  $s_1$  there is a next state where  $p$  and  $q$  both hold.

$M, s_1 \models \neg \text{AG } p$ : It is not the case that in all states on all computation paths from  $s_1$ ,  $p$  holds. (Counter-example: the path  $s_1, s_3, s_4, \dots$ ).

We briefly define satisfaction for CTL formulas inductively on the structure of formulas.

$s \models \top$ : true for any state  $s$ .

$s \models \perp$ : true for no states.

$s \models \text{EX } \phi$ : true if for some directly reachable state  $s'$ ,  $s' \models \phi$ .

$s \models \text{AX } \phi$ : true if for all directly reachable state  $s'$ ,  $s' \models \phi$ .

$s \models \text{EF } \phi$ : true if for some computation path  $\pi = s_0, s_1, s_2, \dots$  with  $s_0 = s$ , some  $s_i \models \phi$ .

$s \models \text{AF } \phi$ : true if for all computation paths  $\pi = s_0, s_1, s_2, \dots$  with  $s_0 = s$ , some  $s_i \models \phi$ .

$s \models \text{EG } \phi$ : true if there exists a computation path  $\pi = s_0, s_1, s_2, \dots$  with  $s_0 = s$  and each  $s_i \models \phi$ .

$s \models \text{AG } \phi$ : true if  $s \models \phi$  and for all computation paths  $\pi = s_0, s_1, s_2, \dots$  with  $s_0 = s$ , each  $s_i \models \phi$ . (One can think of  $\text{AG}$  expressing an invariant).

$s \models E \phi_1 \cup \phi_2$ : true if for some computation path  $\pi = s_0, s_1, s_2, \dots, s = s_0$  and for some  $k \geq 0$  each  $s_j$  with  $j < k$   $s_j \models \phi_1$  and  $s_k \models \phi_2$ .

$s \models A \phi_1 \cup \phi_2$ : true if for all computation paths  $\pi = s_0, s_1, s_2, \dots, s = s_0$  and for some  $k \geq 0$  each  $s_j$  with  $j < k$   $s_j \models \phi_1$  and  $s_k \models \phi_2$ .

$s \models \phi_1 \ \& \ \phi_2$ : true if  $s \models \phi_1$  and  $s \models \phi_2$ .

$s \models \phi_1 \ \vee \ \phi_2$ : true if  $s \models \phi_1$  or  $s \models \phi_2$ .

$s \models \neg\phi$ : true if  $s \not\models \phi$ .

We’ve seen that temporal logics can be useful for expressing properties about the transitional behaviour of systems that would be difficult to formulate using standard Hoare-style specification logics. Next we describe how a model checker can verify CTL properties of Kripke structures automatically.

### Verification

The automatic part of model checking comes in the verification step. The model checker takes as input a model  $M = (S, S_0, R, L)$ , and a specification  $\phi$  and determines if the specification is true of the model.

There are several different techniques for solving the model checking problem. Rather than solving the problem directly, it is often easiest to determine the set of states  $T$  that satisfy a CTL specification. Then the model checking question for the whole system reduces to determining if:  $T \subseteq S_0$ .

The task of associating a set of states with temporal logic specifications can be performed by a simple syntax-directed recursive algorithm. For example, every state satisfies  $\top$ . Similarly, the set of states satisfying  $\phi_1 \vee \phi_2$  is just the union of the sets of states satisfying  $\phi_1$  and  $\phi_2$ . The states satisfying forms containing temporal operators are more difficult to implement, but can be built up using fixed-point computations. An algorithm using this general technique is described in Chapter 3.

Thus, the verification step of model checking can be performed automatically by computing the set of states that satisfy a CTL specification.

### Efficiency and Limitations

Efficient algorithms for local model checking, linear in the size of the model and specification, have been developed using similar techniques. However, they suffer from a key limitation: the algorithm requires an explicit representation of the model. For complex systems, an explicit representation can be too large to store using current resource limitations. In practice, this limits local model checking to fairly small systems and is known as the “state explosion” problem.



A key breakthrough in efficient model checking was made by Ken McMillan [McM93]. His PhD thesis describes a technique for symbolic model checking where the model  $M$  is represented implicitly using Bryant’s ROBDDs [Bry86]. BDDs can be used to avoid the state explosion problem and in practice, often allows far larger systems to be verified. Most recent model checkers have been built using McMillan’s technique (or indeed, based on his SMV system). However in this project, we focus on building a tool with an expressive modelling language rather than on efficient model checking algorithms.

## 1.2 SAFL

SAFL [MS00] is a functional hardware description language (HDL). Descriptions in SAFL are quite different than other description languages such as structural Verilog or VHDL. Instead of specifying the structural properties of hardware circuits, a SAFL programmer only describes the behaviour of the program. An optimising compiler then translates a behavioural SAFL specification to structural Verilog. A complete description of the language is given in [MS00].

The syntax of SAFL programs resembles other functional languages such as ML or Haskell. SAFL expressions have the following abstract syntax (the full SAFL syntax is given in Appendix A):

$$\begin{aligned}
 e ::= & \ c \mid x \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{let } \vec{x} = \vec{e} \text{ in } e_0 \\
 & \ \text{case } e \text{ of } e_1 \Rightarrow m_1 \dots \text{default} \Rightarrow m \\
 & \ \text{a}(e_1, \dots, e_{\text{arity}(\text{a})}) \mid f(e_1, \dots, e_{\text{arity}(f)}) \mid
 \end{aligned}$$

SAFL functions are defined like this:

$$\text{fun } f(\vec{x}) = e$$

And programs as a list of function definitions followed by a main function:

$$\begin{aligned}
 & \text{fun } f_1(\vec{x}_1) = e_1 \\
 & \dots \\
 & \text{fun } f_n(\vec{x}_n) = e_n \\
 & \text{do main}(\vec{x}_n) = e
 \end{aligned}$$

The following SAFL program (due to Sharp) implements a 16-bit add-shift multiplier, and serves as a concrete example of SAFL syntax.

```

fun mult(x:16,y:16,acc:16):16 =
  if (x=16'0 | y=16'0) then acc
  else mult(x << 16'1,y >> 16'1,if y[0] then acc+x else acc)
do
  main(x:16,y:16):16 = mult(x,y,16'0)

```

The syntax  $w'v$  denotes the constant  $v$  with width  $w$  and  $y[0]$  is a 1-bit slice containing the  $0^{th}$  bit of  $y$ .

The key feature of SAFL programs is that they can be statically allocated. That is, the storage required to implement a SAFL description can be determined from the program's structure. Several restrictions are needed to realise this property. First, unlike other functional languages, SAFL is not higher-order; functions may not be used as values. Second, recursive calls are only allowed in tail-call contexts.

In this project, we use SAFL as a modelling language for a model checker. Thus, from a SAFL description of a hardware circuit, we extract a transition system that captures the program's behaviour. This process is described in detail in Chapter 2.

### 1.3 Project Goals

As hinted above, the discrepancy between the actual system and the model of the system can be a practical limitation in verifying systems. Often, features must be omitted in the model – thus the property verified by a model checker is only as correct modulo the assumption that the model correctly captures the behaviour of the system.

The goal of this work is to shrink or eliminate this discrepancy. By using SAFL as a modelling language for a model checker, it is hoped that the process of designing and implementing a hardware system can be integrated with the verification process. Just as programmers in a high-level language annotate programs with types, which are checked at compile time by a static analyser, SAFL hardware designers will be able to annotate their descriptions with CTL specifications that are verified as they are compiled to Verilog. Thus, formal verification becomes more accessible for hardware designers.

### 1.4 Dissertation Organisation

Chapter 2 describes the early, background work that was used to design the SAFL model checker. Chapter 3 describes the implementation of the project. Chapter 4 describes in detail the verification of several smaller SAFL programs, as well some measurements of the performance of our model checker on different sized programs. Chapter 5 evaluates the results lists some ideas for future research.

## Chapter 2

# Preparation

*Perhaps believing in good design is like  
believing in God, it makes you an optimist.*

– Conran Terence

---

Our project proposal (reproduced in Appendix E) articulates the following basic goal for this project: “implementing a model checker with SAFL as the modelling language”. While the requirements to complete this project are fairly unambiguous, several refinements about how to proceed were added to this goal before implementation work began. This chapter describes the preparatory work.

### 2.1 Requirements

One of the most important stages of the software development cycle is the requirements analysis stage. Here we examine the broad features that are required of the project, without worrying about implementation details.

#### 2.1.1 Core Behaviour

The desired behaviour of the SAFL model checker is to take as input a SAFL program and a CTL specification and indicate if the program satisfies the specification.

Achieving this goal involves four distinct phases:

**Parsing:** produce a parse tree from a CTL-annotated SAFL source file.

**Static Analysis:** check that a program syntax tree is legal under a simple type system (width checking for values) and that they satisfy the restrictions for static allocation.

**Extract Kripke structure:** produce a transition system from the syntax tree.

**Model Check:** implement a local model checking algorithm to verify the CTL specifications for the transition system; output result of verification.

### 2.1.2 Code Reuse

The project should reuse front-end code from the SAFL compiler. This is an important requirement because it ensures that the language accepted by the SAFL compiler is same language that is accepted by the model checker. If we were to write a SAFL front-end by hand, we might introduce differences between our version of SAFL and the official version. Further, the SAFL code should be interfaced in a clean fashion so that future bug fixes or SAFL extensions can be handled easily. Thus, the model checker should be practically extensible as the SAFL language matures.

This suggests that the project be written in SML/NJ, the same implementation language of the SAFL compiler.

### 2.1.3 Practical Expressibility

Expressing useful specifications for model checking is already a challenging task because the temporal operators can be difficult to reason about. The SAFL model checker elevates the source language from a simple transition system description language to a functional language. In McMillan's SMV model checker, the coupling between the modelling language and CTL is close. In contrast, in the SAFL model checker, the connection between the source language and CTL formulas is somewhat vague. Thus, some practical indication about how to write SAFL specifications, perhaps by supplying the user with information about the translation, is required.

### 2.1.4 Efficiency

This project does not aim to advance the state-of-the-art in efficient model checking. However, a model checker that cannot cope with non-trivial cases is not particularly useful. Thus, we require a local model checker that can verify systems up to the limitations imposed by state explosion. If efficiency becomes a major factor, we will interface our SAFL-based front-end with a symbolic model checker back-end to allow the verification of larger systems.

### 2.1.5 Extensions

Some of the extensions listed in the project proposal are, upon further consideration, not practical. Here we list a revised set of possible extensions:

**Symbolic model checking:** implement a model checking algorithm that uses an implicit representation of Kripke structures, using OBDDs. Alternatively, implement a translator to another symbolic model checker.

**Counter-examples:** implement an extended model checking algorithm that produces counter-examples when a specification fails.

**Granularity of transition systems:** There are many ways to extract a transition system from a high-level language. We will implement at least one of these techniques for the project. However, it might be useful to compare different approaches.

**Verify SAFL transformations:** The optimising SAFL compiler includes optional transformations to produce more efficient hardware specifications from SAFL programs. Typically these involve a space/time tradeoff. An interesting extension would be to verify that the transformed versions satisfy the same specifications as the original sources.

Our analysis clarifies what is required of a successful project. We will revisit this analysis in Chapter 5.

## 2.2 Further Design

The previous section gives a clear set of requirements for this project. However, the phase that extracts a Kripke structure from a SAFL program is still under-specified. We expand the design of that phase so that its implementation is more easily realisable.

### 2.2.1 Transition Systems

Traditional model checkers verify CTL properties of systems represented by Kripke structures. Accordingly, the modelling language of most model checkers is fairly primitive (*i.e.*, closely resembling Kripke structures). For example, in the SMV system's input language one describes a model by giving the initial values of and a next value relation for states in the system. While many systems can be modelled effectively using SMV, it is often desirable to use a more expressive modelling language in order to simplify the modelling stage of verification. However the transition system implicit in a description written in such a language is not always obvious. Here, we describe a general strategy for inferring a transition relation, and thus a Kripke structure, from a high-level language.

An *operational semantics* defines the precise meaning of expressions and programs in a language. An operational semantics is usually given as a set of syntax-directed rules that define when a more complicated language expression can be reduced into a simpler one. For example the rules in a

small-step semantics for a call-by-name version of the  $\lambda$ -calculus that allow one to reduce a function application to a value (typically an integer or a function closure) go like this:

$$\frac{\langle e_1, s \rangle \Downarrow \langle e'_1, s' \rangle}{\langle e_1 e_2, s \rangle \Downarrow \langle e'_1 e_2, s' \rangle} \quad (\text{APP-CONGRUENCE1})$$

$$\frac{\langle e_2, s \rangle \Downarrow \langle e'_2, s' \rangle}{\langle e_1 e_2, s \rangle \Downarrow \langle e_1 e'_2, s' \rangle} \quad (\text{APP-CONGRUENCE2})$$

$$\langle (\lambda x. e) v, s \rangle \Downarrow \langle e[v/x], s \rangle \quad (\text{APP})$$

The notation  $\langle e, s \rangle \Downarrow \langle v, s' \rangle$  means that expression  $e$ , with state  $s$  (which maps identifiers to their values), can be simplified to value  $v$  with state  $s'$ .

An operational semantics, by providing a model of the execution of programs, gives a straightforward way to extract a Kripke structure from a high level language. Each transition  $\langle e, s_0 \rangle \Downarrow \langle v, s_1 \rangle$  in the semantics is a transition between states  $q_0$  (corresponding to  $e$ ) and  $q_1$  (corresponding to  $v$ ) in the Kripke structure. Each state  $q_i$  is labelled with the values of the variables in the state  $s_i$ . The initial state is the state resulting from the first rule applied.

### 2.2.2 Choosing a Semantics

The previous section showed how a transition system can be extracted from the operational semantics of a high-level description language. If a high-level language is to be used as the modelling language in a model checker, then the choice of semantics for that language becomes critical. The semantics chosen should reflect the granularity of the transitions that are of concern in the system. If the transitions in the semantics are too coarse, then the model will fail to capture important aspects of the system. On the other hand, a model that includes too much detail may be too large to verify efficiently.

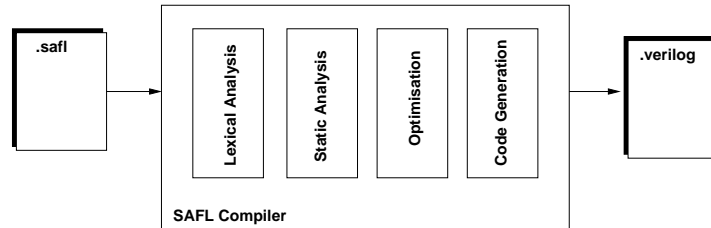


Figure 2.1: SAFL Compiler Stages

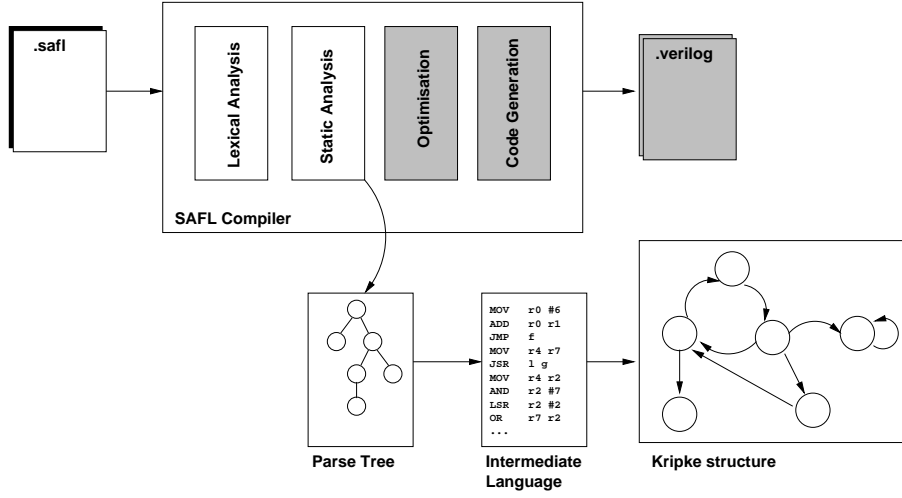


Figure 2.2: Design diagram for extracting Kripke structure from SAFL descriptions

### 2.2.3 Extracting Kripke Structures from SAFL Programs

The SAFL language is implemented by a compiler that translates behavioural specifications to structural Verilog. The compiler has the phases of a traditional compiler: lexical analysis, type checking, optimisations, and code generation, as shown in Figure 2.1

Given this existing implementation, there are several options for extracting a transition system from SAFL programs. We could use the source language directly, with a semantics extended from the standard call-by-value  $\lambda$ -calculus semantics. Alternatively, we could define a semantics for the Verilog subset that the SAFL compiler targets, and extract the transition system from the output of the compiler. Thirdly, we could translate the source language to an intermediate language, and use the semantics of the intermediate language to get our transition relation. Neither of these choices is clearly superior to the others. The first option gives the most flexibility in the granularity of the semantics (because we reuse almost no code from the SAFL implementation). The second allows us to use the actual output of the SAFL compiler. However, it leaves us susceptible to possible bugs (a concern in a compiler built as part of an uncompleted PhD!). In the end, we chose the final option that gives us some control over the granularity of the transition system while still allowing us to reuse the front-end of the SAFL compiler.

The intermediate language, SAFLASM, is extended from a similar language by Mycroft and Sharp given in [MS00]. It closely resembles the features available on modern instruction set architectures. Because the intermediate language is very primitive (unlike the high-level SAFL description),

its semantics is straight-forward and fairly unambiguous. In particular, we have little choice over the granularity of transitions in the semantics. Each instruction is processed in a single step – the variables changed at each step can be statically determined directly from the syntax. The final step going from the intermediate language to a transition system can be achieved by a simple interpreter that steps through the instruction sequence for each possible set of values of the free variables at the start of the sequence.

Thus, the design for the front-end of the model checker reuses the lexical and static analysis of the SAFL compiler (with minor modifications to handle the input of CTL specifications). This yields a parse tree that is then translated to the intermediate language using an extension of the translation of Mycroft and Sharp. Finally, an interpreter is run over the instructions to build the explicitly represented Kripke structure. Finally, the local model checker can be run on this structure to complete the verification. Figure 2.2 shows a diagram of the extraction process.

The verification stage of the model checker itself uses standard local model checking algorithms, as discussed in Chapter 1. Thus, no further refinements to that portion of the design are needed.

## 2.3 Programming Practices

The project is implemented in SML/NJ, the same language that is used to implement the SAFL compiler. SML is well-suited to language and symbolic manipulation projects with features including user-defined data types and pattern matching. Additionally, it is strongly typed; its expressive type system catches many errors at compile time without being too restrictive.

The design of the project naturally divides into several distinct phases: lexing, parsing, static analysis, extraction of a Kripke structure, and verification. SML’s module system is used to encapsulate the implementation details each of the phases of the model checker. Only the public signature of certain values are allowed to escape from modules. Additionally, type definitions are separated from functions that operate on them. For example, the **Static** module includes routines to check properties of **AbSyn** parse trees. Similarly, the **Compile** module translates **Lb1Syn** (labelled syntax trees) into **SAFLasmSyn**.

The development environment used is **emacs** with **sml-mode**. **CVS**<sup>1</sup> is used for revision control, and **unison**<sup>2</sup> for regular remote backup.

---

<sup>1</sup><http://www.cvshome.org>

<sup>2</sup><http://www.cis.upenn.edu/~bcpierce/unison/>



## Chapter 3

# Implementation

*Many things difficult to design prove easy to performance.*

– Samuel Johnson

---

The SAFL model checker is implemented as a series of passes over a syntax tree. First, the SAFL compiler front-end is used to produce a parse tree. Then the tree is translated to the SAFLASM intermediate language (closely resembling RISC assembly code). Next, a Kripke structure is extracted from this intermediate language by an interpreter. Finally, the verification step is performed by a local model checker. This chapter describes the implementation of each of these steps in detail. We also describe two extensions: a function to produce counter examples for failed specifications, and a translation from SAFLASM to SMV.

### 3.1 Front End

The front end of the SAFL model checker is adapted from the SAFL compiler. There are three phases that together produce a legal SAFL parse tree from a source file: the lexer, the parser, and the static analyser. All three of these are modified so that CTL specifications can be included in SAFL programs. A SAFL program for model checking (a full grammar is given in [Appendix A](#)) is a list of function definitions, followed by the `do` keyword, and a `main` function that specifies the input and output variables for the system. CTL specifications are enclosed in a special comment, as in the following program (if a program does not include a specification, the trivially true CTL formula `true` is inserted automatically):

```
fun f() : 1 = ...  
fun g(x:5) : 3 = ...
```

```

do
(** spec: AG ! c **)
main(x:5) : 3 = (f(); g(x))

```

The addition of CTL annotations to SAFL programs requires a few small changes to the front-end. The lexer is extended to recognise CTL tokens (`EX`, `AX`, `AF`, ...). A `ctl` datatype is added to the data structures used to represent syntax trees. The parser is extended with rules to parse CTL specifications for programs, as described above. Finally, the static analyser, is modified to ignore the CTL components of programs during its analysis.

Two other modifications were made to the SAFL front-end. First, the parsing of `case` statements was changed so that a `default` case is always required. Without a default, the behaviour of `case` is not always well-defined. Consider the following program:

```

fun odd(x:2) : 1 =
  case x of
    2'1 => 1'1
    2'2 => 1'0
    2'3 => 1'1
do
  odd(2'0)

```

It takes a 2-bit value and returns 1 if the value is odd. However, it omits the case for 0. Thus, even though the program is well formed according to the static rules, the behaviour of the expression `odd(2'0)` is undefined.<sup>1</sup> Rather than having a special case for `NoCaseMatch` in the Kripke structure extracted from programs like this, we chose to insist that all `case` statements have a default case. (This also allows us to treat `case` statements as syntactic sugar that can be encoded using conditionals).

Second, SAFL allows `external` function definitions. These are useful for interfacing other hardware devices with SAFL programs. For example, a 32-bit CPU written in SAFL might include the following definition for memory I/O:<sup>2</sup>

```
external mem_acc (address:16,data:16,write:1):16
```

For the standard SAFL compiler, such definitions pose no special problems. The compiler simply leaves wires that can be connected to other hardware devices with the same interface. However, for model checking, we require that the behaviour of the system be well defined. Thus, we do not allow `external` declarations.

---

<sup>1</sup>Sharp's SAFL interpreter raises a `NoCaseMatch` exception.

<sup>2</sup>This example from a SAFL description of a stack processor by Sharp

### 3.1.1 Translate Equivalent Forms

Before performing the translation to the intermediate language, we translate some more complicated forms to other semantically equivalent forms. Most meta-language tools (translators, type checkers, verifiers) have a case to handle each form in the language. It is often desirable to translate away forms that can be derived from more primitive ones to reduce the number and complexity of cases which must be handled.

#### Case Statements

case statements have the form:

```
case expr of
  match_1 => e_1
  ...
  match_n => e_n
  default => e_default
```

Informally, the semantics of case statements is as follows. First, `expr` is evaluated. If any `match_i` is equal to its value (checked in order of declaration; that is, if two `match_i` are equal to the value of `expr`, the first one in the match list is used), then the value of the entire expression is `e_i`. Otherwise, if no cases are matched, the value is `e_default`.

We translate case statements into a `let` declaration and nested conditionals as follows:

```
let new_var = expr in
  if (new_var = match_1) then e_1
  ...
  else if (new_var = match_n) then e_n
  else e_default
```

In this translation, `new_var` is a fresh variable name generated by the compiler.

#### Joins

Two expressions `e1` and `e2` with widths `w1` and `w2` can be joined using `join` operator. This results in an expression whose length is `w1+w2` and whose bits are the concatenation of the value of `e1` with that of `e2`. Joins can be eliminated by a translation using the primitive operations of shifting left and binary OR. The simple translation goes like this (note that `w2` is a constant computed by compiler):

```
(e1 << w2) | e2
```

### Slices

Bit slices are also translated into other more primitive operations on binary numbers. A bit slice has the form:

`expr[from,to]`

and the value of the `from` through `to` bits of `expr` (inclusive). For example, if `expr` has the value 10001110 the expression `expr[5,1]` has value 00111.

Bit slices can be eliminated by masking out the desired bits, then shifting them right. Translation of `expr[from,to]` goes like this. First, compute the following constants:

Pre-compute slice width `sw = (from - to + 1)`

Pre-compute mask `m = (2sw - 1)`

Pre-compute shifted mask:

`sm = if (to = 0) then m else m << (to - 1)`

Then produce the translation (if `to = 0`, can omit the final shift):

`((expr & sm) >> to)`

These translations eliminate complicated, specialised syntactic forms in favour of ones that can be evaluated more easily in the intermediate language.

## 3.2 Translation to Intermediate Language

In this section, we present the translation from SAFL to an intermediate language, SAFLASM. The translation is adapted from Mycroft and Sharp's translation given in [MS00].

This portion of the project implementation is very similar to the code generation phase of most compilers. However, unlike most languages SAFL programs can be statically allocated. Thus, the translator does not use any dynamically allocated storage space (stack or heap). Instead, it uses a (potentially large) finite set of registers.

### SAFLASM Syntax

SAFLASM closely resembles the primitive operations that are available on most modern instruction set architectures. Notable omissions are any instructions for accessing memory (all storage is statically allocated). The syntax and informal semantics are as follows:

label:	MOV	op1 <- op2	Copy op2 to op1
label:	JMP	lb1	Jump to instruction with label lb1
label:	JSR	lb11 lb12	Jump to subroutine at lb11; save return label in lb12
label:	RET	lb1	Return to label stored in lb1
label:	BEQZ	op lb1	If op is zero, jump to lb1
label:	PRIMOP	op1 <- op2 OP op3	Assign op1 <- op2 OP op3
label:	PRIMOP	op1 <- op2	Assign op1 <- OP op2

Primitive operations include arithmetic operations `ADD`, `SUB`, `MULT`, *etc.*, bit operations `AND`, `OR`, `XOR`, `NOT`, *etc.* logical operations that take operands of width 1: `LAND`, `LOR`, `LNOR`, *etc.* Appendix B contains the full language syntax and operational semantics.

### Labelling

While SAFL programs can be statically allocated, the static storage locations are not always immediately obvious from the syntax. In order to provide hints to the compiler about storage locations for expressions, it is helpful to label declarations and expressions with locations.

In the labelling phase, the compiler recurses through the syntax tree, annotating each expression, let-bound variable declaration, and function formal parameter, with a label – a string denoting the location that the expression’s value will be stored in when compiled. For example, the expression `(e1; e2)` might be labelled with locations `r31` and `r32` like this:

`(<r31>e1; <r32>e2)}`

The labels used, as long as they are distinct, do not affect the correctness of the code generation. The labelling used simply increments by one at each expression or variable declaration that it encounters in a recursive descent. Thus, sub-expressions are not necessarily related in any mathematically meaningful way.<sup>3</sup>

### Translation

Our translation to SAFLASM is adapted from a translation given in [MS00]. However, Mycroft and Sharp’s version included a `PAR` command, implementing fork-join parallelism, and instructions to manipulate semaphores for mutual exclusion. With these additional instructions, their translation evaluates non-conflicting arguments to function calls, and let declarations in parallel (expressions do not conflict if the functions called in evaluating them

---

<sup>3</sup>A more clever labelling algorithm might produce labels such that the labels for sub-expressions are related by some mathematical formula. However, the labels tend to be generated by techniques similar to Gödel numberings – and thus, can get quite long.

are distinct). Conflicting calls have to wait for the semaphore to become free to access to the shared resource (in this case, access to a function).

However, evaluating expressions sequentially makes no semantic difference to the underlying transition system. The SAFL source language does not include explicit parallelism. Thus, the programmer has no guarantee that a particular expression will be evaluated in parallel or sequentially with another. As long as our model captures the semantics of SAFL source programs accurately, the specific modelling approach doesn't matter much. Thus, we eliminate the instructions that support concurrency in our translation. Where the Mycroft translation evaluates expressions in parallel, we do them sequentially. This simplification makes no difference to the SAFL programmer because there are no parallel constructs in the language. If SAFL supported explicit, programmer specified parallelism, then it would be important for the extracted transition system to model concurrency. However, as the concurrency in SAFL is automatically inserted by the compiler, modelling the concurrency would only be useful to verify that the SAFL compiler is implemented correctly. Assuming that the analysis implemented in the SAFL compiler is correct, there is no need to model concurrency until the language supports explicit parallel constructs.

The translation,  $\llbracket \cdot \rrbracket$  from SAFL to SAFLASM is given inductively on the structure of language forms in SAFL. The complete translation is described in Appendix B. Here we give some of the illustrative cases. The translation uses the labelled locations of expressions as storage for compiled sub-expressions. Thus, in the translation of an expression, we are given a location to store the value of that expression. For example, compiling a constant  $c$  to storage location  $l$  goes like this:

$$\llbracket c \rrbracket l = \text{MOV } l \leftarrow c$$

Similarly, a variable expression  $x$  is compiled to  $l$  with the following code:

$$\llbracket x \rrbracket l = \text{MOV } l \leftarrow M_x$$

The notation  $M_x$  denotes the current location that holds the value of  $x$ . The variable  $x$  might be a  $\lambda$  or let-bound variable. Thus, we maintain a variable environment that maps variables to labelled locations to keep track of variable locations.

Conditionals require a slightly more complicated translation.

$$\begin{aligned} \llbracket \text{if}(\langle l_1 \rangle : e_1) \text{ then } e_2 \text{ else } e_3 \rrbracket l = & \quad \llbracket e_1 \rrbracket l_1 \\ & \text{BEQZ } l_1 \text{ lfalse} \\ & \llbracket e_2 \rrbracket l \\ & \text{JMP } l_{\text{next}} \\ \text{lfalse} : & \quad \llbracket e_3 \rrbracket l \\ \text{lnext} : & \quad \dots \end{aligned}$$

First, we recursively compile the guard of the conditional to its labelled location,  $l_1$ . Then we test if its value is equal to zero and branch accordingly. The true branches  $e_2$  is compiled to a list of instructions. Following these, we insert a **JMP** statement to skip the instructions implementing the false branch.

Recursive function calls are easy to compile because of the restriction to tail-calls. Thus, we can use a common tail-call optimisation, updating the registers holding the arguments for the function call and jumping to the function's entry point:

$$\begin{aligned} \llbracket f(\langle l_1 \rangle e_1, \dots \langle l_n \rangle e_n) \rrbracket l &= \llbracket e_1 \rrbracket l_1 \\ &\dots \\ &\llbracket e_n \rrbracket l_n \\ &\text{MOV } Mformals_1 \leftarrow l_1 \\ &\dots \\ &\text{MOV } Mformals_n \leftarrow l_n \\ &\text{JMP Entry}_f \end{aligned}$$

Function definitions are compiled by compiling their bodies to the storage location  $\text{Res}_f$ :

$$\llbracket f(x_1 : w_1, \dots x_n : w_n) : w = e \rrbracket = \text{Entry}_f : \llbracket e \rrbracket \text{Res}_f \\ \text{RET } L_f$$

Finally, programs are compiled by compiling each function definition and the body of **main**. Finally, an instruction of the form:

**1b1: JMP 1b1**

is emitted (trapped during interpretation as halting).

The remaining cases in the translation are contained in the appendix. Here we give a brief example of the compilation to SAFLASM of a simple expression:

**if (f(x)) then 3'0 else g(y[3,1])**

In this example, variable names are unique, so we use them as their own storage locations (*i.e.*,  $M_x = x$ ). Assume that the storage location of **f**'s variable is **fvar** and **gvar** for **g**'s. Then the code generated by compiling this expression to a 3-bit register **res** is:

```
line_2:      MOV      fvar <- x
line_3:      JSR      L_f, Entry_f      !f(x)
line_4:      MOV      r4 <- Res_f
line_5:      BEQZ     r4 line_8          !guard
line_6:      MOV      res <- 3'000      !true branch
line_7:      JMP      line_16
```

```

line_8:      MOV      r6 <- y           !false branch
line_9:      MOV      r7 <- 6'001110
line_10:     AND.6    r9 <- r6, r7
line_11:     MOV      r8 <- 6'000001
line_12:     RSHIFT   r10 <- r9, r8     !y[0]
line_13:     MOV      gvar <- r10
line_14:     JSR      L_g, Entry_g      !g(x)
line_15:     MOV      res <- Res_g
line_16:     JMP      line_16

```

### 3.2.1 Optimisation

The above translation is naïve in its use of resources. Specifically, it uses far more registers than are needed. We could perform liveness analysis on the naïve compiled code and then register allocation by colouring to reduce the number of registers needed. However, while this would improve the space efficiency of compiled SAFL programs, it makes specification much more difficult. In the naïve implementation, it is easy to identify registers with the values that they hold. If registers are reused, this easy to understand connection between the source language and the compiled version would be lost. Finally, the size of Kripke structure is determined mostly by the number and width of the inputs to `main` (as is explained in detail in the next section); the large number of registers doesn't affect the size of the structure much.

## 3.3 Building the Kripke Structure

Section 2.2.1 described a method for extracting a transition system from a high-level language. Using the steps in an operational semantics, a notion of state ( $S$ ) and transitions between states ( $R$ ) can be inferred. The values of the variables at each state are the atoms that are true in that state ( $L$ ). From this, it is easy to see how to construct a Kripke structure  $M = (S, S_0, R, L)$  from a semantics.

Our implementation uses an interpreter that implements the semantics in Section B.2 to produce execution traces of SAFLASM programs. First, we augment the SAFLASM code with two special registers: `pc` which tracks the interpreter's current location in the program, and `c` to indicate arithmetic overflow. Then we create execution traces of the program for different input values to `main`. Each instruction modifies `pc`, and possibly some other registers. For example, if the current variable environment is `env`, executing the instruction

```
lbl:      JMP lbl_n
```



results in a new environment  $env'$  updates `pc`'s value to `lbl_n`. Similarly, executing

```
lbl_n:   ADD r_i r_j 3'7
lbl_n+1: ...
```

results in a new environment where `r_i` is assigned the value `r_j + 3'7`, and `pc` is assigned `lbl_n+1`. Additionally, if `r_i` does not have enough bits to store the value `r_j + 3'7`, `c` is updated to `1'1`. Other instructions are handled similarly, according to the semantics given in Section B.2.

The interpreter detects when it has reaches an instruction that it has already processed with the same variable environment and halts. Because of the translation used, all synthesised SAFLASM programs have infinite looping: either by non-terminating recursion or else by the

```
lbl: JMP lbl
```

instruction that follows the compilation of the body of `main`.

Once the execution traces are complete (resulting in a list of environments mapping registers to values in their order of execution), building a Kripke structure is trivial. Each unique environment (mapping of registers to values) results in a state in  $S$ .  $R(s, s')$  holds if the environment represented by  $s$  is followed in the execution trace by the environment represented by  $s'$ . The labelling function,  $L(s)$  is determined by the values of each register in the environment for  $s$ . The single start state corresponds to a pre-environment where `pc` equals the first instruction in the program, and the values of the other registers are undetermined, denoted by  $?$ .

### 3.3.1 Non-Determinism and Branching

The simple semantics of SAFLASM programs results in completely deterministic execution traces. For a particular valuation the registers, the execution trace can be unwound to a single chain of states. Generally, branching in a model occurs when the system being models exhibits non-deterministic behaviour. However, the only non-determinism in SAFL programs is in the inputs supplied to the `main` function. Thus, the only branching in synthesised Kripke structures is from the initial state, corresponding to possible input values to `main`. When building Kripke structures from SAFLASM programs, the interpreter automatically produces an execution trace for each input value. For example, the Kripke structure created from the following simple program:

```
fun loop(x:2):1 =
  loop(x)
do
  main(x:2):1 = loop(x)
```

has four possible execution traces from the initial state: one for each of the possible values of the 2-bit input  $\mathbf{x}$ . The interpreter simulates the execution of each possible input to `main`. Thus, the synthesised Kripke structures typically have a form similar to the one shown in Figure 3.1.

In this section, we've shown how the SAFL model checker extracts Kripke structures from SAFL source programs by compilation to SAFLASM, then interpreting the low-level code. Now we consider the verification algorithm.

### 3.4 Verification

The final phase in the model checker performs the actual verification. There are many different model checking algorithms that can be used to verify that a Kripke structure satisfies a CTL specification. One simple approach is to compute the set of states that satisfy the formula directly. The states satisfied by a CTL form  $\phi$  can be characterised by a predicate *sat* that computes the set  $\{s \mid s, M \models \phi\}$ . Rather than give a full description of the algorithm, we highlight a few of the interesting cases. The complete definition of *sat* is given in Appendix C. The algorithm here is adapted from ones in [JGP99, HR00]:

Many of the cases can be handled by building the set directly, using simple set operations.

$$\begin{aligned} \text{sat}(\top) &= S \\ \text{sat}(\perp) &= \emptyset \\ \text{sat}(p) &= \{s \mid s \in S \ \& \ s \in L(p)\} \\ \text{sat}(\neg\phi) &= S - \text{sat}(\phi) \\ \text{sat}(\phi_1 \vee \phi_2) &= \text{sat}(\phi_1) \cup \text{sat}(\phi_2) \\ \text{sat}(\phi_1 \ \& \ \phi_2) &= \text{sat}(\phi_1) \cap \text{sat}(\phi_2) \\ \text{sat}(\text{EX } \phi) &= \{s \mid s \in S \ \& \ \exists s'. (R(s, s') \ \& \ s' \in \text{sat}(\phi))\} \end{aligned}$$

Each state in  $S$  satisfies  $\top$ , while no states ( $\emptyset$ ) satisfy  $\perp$ . The states satisfying an atomic proposition  $p$  are those that are labelled by  $p$ . The set of states satisfying  $\neg\phi$  is the complement of the states in  $S$  that satisfy  $\phi$ . The cases for  $\&$  and  $\vee$  are implemented by set intersection and union respectively. The states satisfying  $\text{EX } \phi$  are the ones that have a transition in  $R$  to a state satisfying  $\phi$ .

Other cases can be handled by translation to other CTL forms. For example,

$$\begin{aligned} \text{sat}(\text{EF } \phi) &= \text{sat}(\text{E } \top \text{ U } \phi) \\ \text{sat}(\text{AX } \phi) &= \text{sat}(\neg \text{EX } \neg\phi) \end{aligned}$$

The translation of  $\text{EF}$  makes sense according to informal CTL semantics: there exists a future state where  $\phi$  holds if there exists a state where  $\top$  holds until  $\phi$  holds.  $\top$  is true of all states. Thus, the two expressions are both precisely true of states where  $\phi$  holds in the future (or possibly the

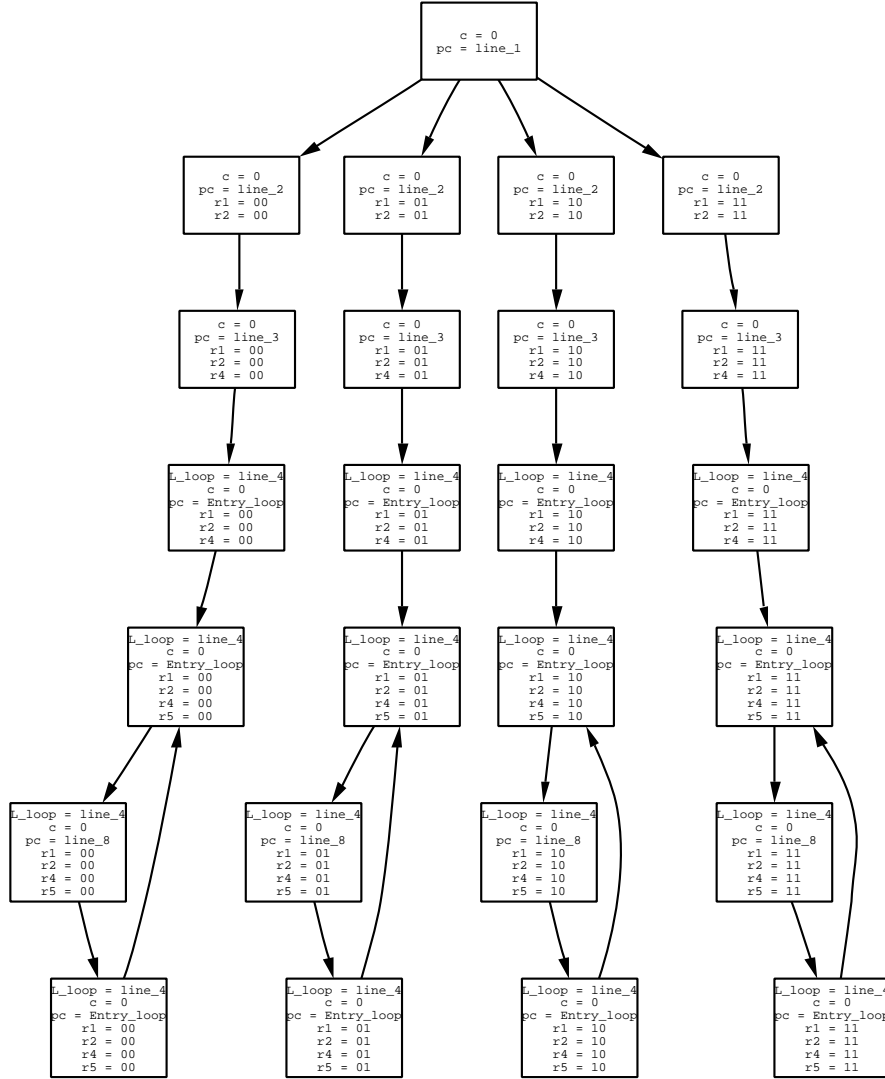


Figure 3.1: Typical Kripke structure for a SAFLASM program

present – recall that the future includes the present in the standard CTL interpretation). Similarly, AX can be encoded using EX: a state is in AX  $\phi$  if there are no successor states in  $\neg\phi$ .

The solutions for other cases of *sat* are computed by fixed-point computations. For example,  $E \phi_1 \cup \phi_2$  is the solution to this fixed-point equation:

$$sat(E \phi_1 \cup \phi_2) = \mu Z. \phi_2 \vee (\phi_1 \& EX Z)$$

Again, stepping through this formula, we see that the set of states that satisfies  $E \phi_1 \cup \phi_2$  – states in a computation path where  $\phi_1$  holds until  $\phi_2$  – are the ones where either  $\phi_2$  holds or  $\phi_1$  and  $Z$  holds in a successor state. A simple least-fixed point computation can be used to compute these sets. We start with  $Z = \emptyset$ , and repeatedly assign  $Z = (\phi_2 \vee (\phi_1 \& EX Z))$  on each iteration until  $Z$  stabilises.

The core of a local model checker is an algorithm is an inductive algorithm that computes the states which satisfy a CTL specification. The cases are handled by set operations, translation to other CTL forms, or fixed point computations.

### 3.5 Producing Counter Examples

One important feature of most model checkers is that they are able to automatically produce a path illustrating a counter example when a specification is false. As an extension to the basic project, we implemented a function to generate counter examples for CTL specifications.

Any path in the Kripke structure where the specification fails to hold can serve as a counter example. For example, if the specification is of the form  $AG \phi$ , then finding a counter example just means finding a single computation path in the system where  $\neg\phi$  holds. However, if the specification is of the form  $EF\phi$ , then it must be true that on all paths,  $\neg\phi$  holds. Thus, we cannot produce a single counter example for false specifications that contain a  $E$  path quantifier – the set of all paths serves as a collective counter example. The fact that the counter example for a specification with a  $A$  path quantifier is the witness for a specification with a  $E$  is noticed by Clarke et al. [JGP99].

Thus, we can reduce the problem of finding computation paths to serve as counter examples to some trivial cases ( $\neg\phi$ , atomic propositions, etc.) and to finding witnesses for EX, EG, and EU. The easiest, finding a witness for  $EX\phi$  from state  $s$  is any path  $s \rightarrow s'$  where  $R(s, s')$  and  $s' \in \phi$ .

A witness for  $EG \phi$  involves several iterations. We start with the singleton path,  $s$ , and repeatedly add successors  $s'$  to the last processed state where  $s' \models \phi$ . We halt when we reach a state already in the witness path. If we fail to find a complete path satisfying  $EG \phi$  (by reaching a state where no

successors satisfy  $\phi$ ), then we backtrack to the last state in the path where we had a choice of successors satisfying  $\phi$ , and restart the algorithm.

Similarly, witnesses for  $E \phi_1 \cup \phi_2$  can be found by repeatedly adding successors in  $R$  of states that satisfy  $\phi_1$ . If a successor satisfies  $\phi_2$ , then the path is complete.

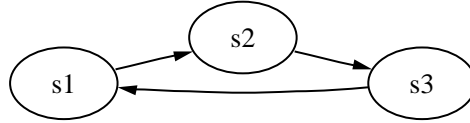
Because we cache the results of model checking CTL specifications, and counter examples are only generated after the model checker fails to verify a formula, the states satisfying all of the formulas and sub-formulas in a specification are already be computed and stored in the cache.

### 3.6 Translation to SMV

The local model checking algorithm described above requires an explicitly defined model in order to verify specifications. Other model checkers, such as SMV operate on symbolically defined models. Thus, the model can be implicitly defined in a high level language, avoiding the state explosion problem in some cases. Here we describe a translation of SAFLASM to the SMV input language. This allows SAFL to be used as the modelling language for an industrial-grade model checker.

#### 3.6.1 SMV Syntax

The SMV input language is a robust language with many features. However, our translation only uses a few forms. An SMV module definition has three sections: **VAR**, **ASSIGN**, and **SPEC**. The **VAR** section contains variable declarations. The **ASSIGN** section describes the transition behaviour of the system. It includes **init** declarations, that describe the initial values of variables in the system and **next** declarations that define the next values of variables. For example, the transition behaviour of the following system:



can be encoded in SMV using a single variable  $v$  with enumerated type

```
{s1, s2, s3}
```

whose corresponding **ASSIGN** declarations are:

```

init(v) := s1
next(v) :=
  case
    (v = s1) : s2
    (v = s2) : s3
  
```

```

    (v = s3) : s1
  esac;

```

Finally, the **SPEC** section lists a CTL property to be verified.

The translation from SAFLASM to SMV's input language is quite simple. First, **VAR** declarations are generated for each register. These have type `array w..0 of boolean` where `w` is the width of the register. Registers that are used to hold labels rather than numerical values have an enumerated type composed of all each instruction label in the program. A single **init** declaration assigns the first label in the program to `pc`.

The trickiest part of the translation is in producing the `next(reg)` declarations for each register. Each instruction modifies at least one register. The registers modified at each instruction can be determined by analysing the SAFLASM syntax tree. For example if the only instruction to modify a 3-bit register `r1` is

```

1b1 :    MOV r1, 3'7

```

then we can define `next(r1)` as

```

next(r1) =
  case
    (pc = 1b1) : [1, 1, 1];
    (1)        : r1;
  esac;

```

Using this technique, it's easy to see how we build up the set of instructions that modify each register and produce the correct `next` declaration. Finally, CTL specifications are output in syntax used by SMV. Appendix D contains an example of a translation to SMV for a SAFL program.

### 3.7 Modules

We close this chapter by listing the modules produced for the project. Note that **Parse** and **Static** are largely due to Sharp.

```

Parse.parse           : string -> AbSyn.prog
Static.check          : AbSyn.prog -> AbSyn.prog
Label.label_prog      : AbSyn.prog -> LblSyn.prog
Compile.compile_prog  : LblSyn.prog -> SAFLasmSyn.code
Kripke.code2kripke    : SAFLasmSyn.code
                      -> KripkeSyn.kripke
ModelChecker.sat      : KripkeSyn.kripke -> CtlSynctl
                      -> KripkeSyn.StringSet
SMVTrans              : SAFLasmSyn.code -> CtlSynctl
                      -> SMVSyn.smv

```

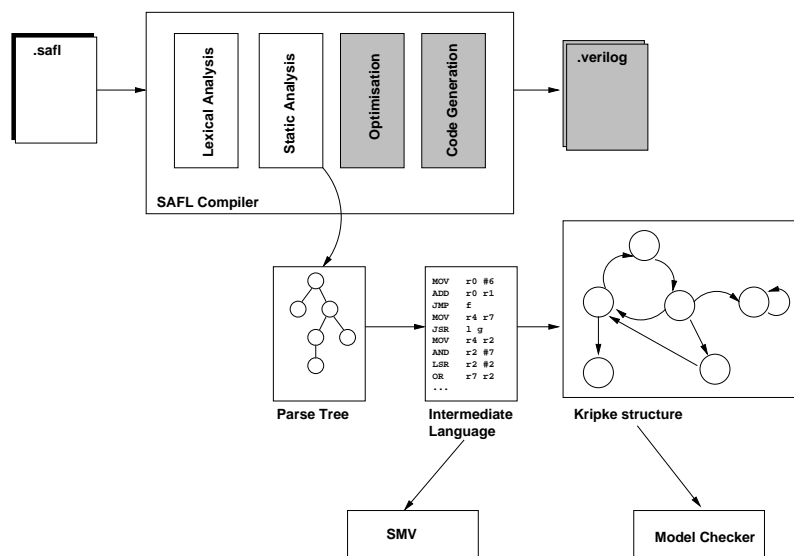


Figure 3.2: Final design for SAFL Model Checker

Additionally, two modules were built for debugging and output:

```

PrettyPrint
Graph

```

The first can be used to print any of the data structures above. The second takes a `KripkeSyn.kripke` and produces a graph description file suitable for use with the graph drawing tools of `graphviz`<sup>4</sup> project (Figure 3.1 was produced using this module). The final design diagram for our implementation is shown in Figure 3.2.

<sup>4</sup><http://www.research.att.com/sw/tools/graphviz/>





## Chapter 4

# Evaluation

*Does it contain any abstract reasoning concerning quantity or number? No. Does it contain any experimental reasoning, concerning matter of fact and existence? No. Commit it then to the flames: for it can contain nothing but sophistry and illusion.*

—David Hume

---

In this chapter, we describe the results of our project with empirical analysis of its efficiency, and with some test runs on sample programs. These results give a sense of the capabilities and limits of the project in practice.

### 4.1 State Explosion

Most explicit state model checkers are limited by the state explosion problem. Each bit whose value is free in the system doubles the state space of the underlying Kripke structure. Thus, the size of Kripke structures grows exponentially. In SAFLASM programs, though the number of registers can be large (because of the unoptimised compilation algorithm) the number of unconstrained input bits is limited to the inputs to `main`. Still, state explosion limits the size of SAFL programs that can be verified using our tool.

In order to quantify the effects of state explosion on verification in practice, we used our implementation to build Kripke structures for an  $n$ -bit add-shift multiplier for several values of  $n$ . The multiplier is again due to Sharp:

```
fun mult(x:n,y:n,acc:n):n =
```

bits	time (ms)	states
1	13	65
2	15	77
4	76	305
8	4548	4865
9	8570	9729
10	100371	19457
11	318824	38913
12	750329	77825

Figure 4.1: Data for n-bit multiplier Kripke structure

```

if (x=n'0 | y=n'0) then acc
else mult(x << n'1, y >> n'1, if y[0] then acc+x else acc)
do main(input:n):n = mult(input, input,n'0)

```

By increasing `n`, the number of possible values of `input` increases. Thus, the interpreter that builds Kripke structures from SAFLASM programs must produce more execution traces, and the size of the resulting Kripke structure grows. The results of our test,<sup>1</sup> shown in Figures 4.1 and 4.2 show that the state explosion problem is indeed a limitation for our model checker. The actual limit on the number of free bits in a SAFL program depends on the program in question. However, at approximately 300,000 states (corresponding to about 16 free bits) the resources of our test setup were exhausted.

This limitation can be overcome by tweaking the size and number of inputs to `main`. Limiting the number of input bits to a SAFL program usually does not affect the transition behaviour of the system. Thus, in a large system, the `main` interface can be used to control the size of the Kripke structure model of the system and possibly avoid the limitations imposed by the state explosion problem. The technique of eliminating unnecessary features from the model of a system is known in the model checking literature as *abstraction*. It is convenient that abstraction can be applied to SAFL programs simply by modifying the inputs and expression in `main`. As an example of abstraction, in the multiplier above, the single formal parameter to `main`, `x`, is passed twice as the argument to `mult`. Thus, while every possible input to `mult` is not modelled in the Kripke structure, a large number of inputs are included. Alternatively, if the model is too large to be verified using explicit state representations and algorithms, the translation to SMV can be used. As SMV does not build an explicit Kripke structure, the state explosion can be avoided in many cases (though not all boolean functions can be represented efficiently using BDDs – some models will still explode).

---

<sup>1</sup>performed on an AMD Athlon 1.0GHz PC with 256MB of RAM and 128MB of virtual memory, running Debian Linux and using SML/NJ 110.0.3.

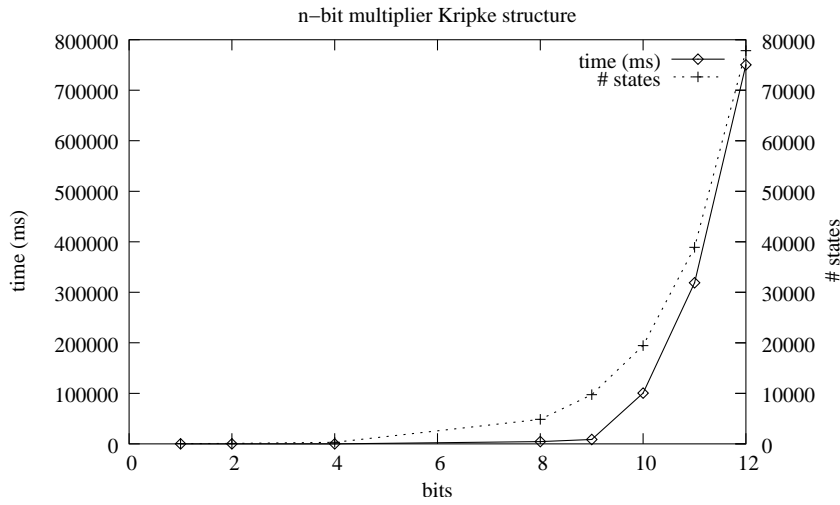


Figure 4.2: Graph for n-bit multiplier Kripke structure

## 4.2 Verifying Programs

In this section we give some examples of SAFL programs and correctness specifications in CTL. We also give the output of running our model checker on these examples.

### 4.2.1 Simple Addition

We start with a simple program to add two integers and a simple correctness specification.

```
fun add(x:3,y:3):3 =
  x+y
do
  (** spec: AG c <> 1'1 **)
  main(x:3,y:3):3 = add(x,y)
```

In plain English, this specification reads: “On all computation paths, it is globally true (AG) that register *c* does not equal the single-bit value 1”. The *c* register is set by the interpreter whenever a primitive operation overflows. Thus, this specification is true of programs where overflow does not occur. However, running this program through our model checker,

```
- Main.go2 "add1.lang";
[parsing started add1.lang]
[parsing completed 53ms]
[doing static checks checking add1.lang]
[labelling parse tree add1.lang]
```

```

[compiling labelled tree add1.lang]
[generating Kripke structure add1.lang ... states: 641]
[elapsed time: 212ms]
[verifying specification...]
CTL cache hit rate: 0.0%
AG (c <> 1'1): no
[finished add1.lang in 798ms]

```

we see that the model does not meet the specification (we delay demonstrating counter examples for failed specifications until the conclusion of this chapter). In particular, if the values of  $x$  and  $y$  add up to an integer greater than 7, then their sum cannot be stored in a three bit register. The problem with this implementation is that the SAFL primitive  $+$  takes an  $n$ -bit by  $n$ -bit binary number and returns an  $n$ -bit binary number. However, if the sum of the two inputs is greater than  $2^n - 1$ , then the result cannot be stored in  $n$ -bits, and overflow occurs. We can fix this, by adding an extra bit to the body of `add` to store the carry out bit of the addition. Our second attempt at an implementation goes like this:

```

fun add(x:3, y:3):4 =
  join (1'0, x) + join (1'0, y)
do
  (** spec: AG c <> 1'1 **)
  main(x:3, y:3):4 = add(x,y)

```

Here we pad the inputs  $x$  and  $y$  to `add` with an extra bit. Thus, the  $+$  primitive operation works on 4-bit inputs even though the actual inputs will always be expressible in three bits. Thus, overflow is avoided for all inputs. Now our model checker verifies the specification:

```

...
[generating Kripke structure add2.lang ... states: 1665]
[elapsed time: 1140ms]
[verifying specification...]
CTL cache hit rate: 0.0%
AG (c <> 1'1): yes
[finished add2.lang in 1779ms]

```

### 4.2.2 Counter

Next we consider an  $n$ -bit counter. It repeatedly calls itself, incrementing an internal register at each iteration. The source for a 3-bit counter goes like this:

```

fun counter(x:3):3 =
  if (x = 3'7) then counter(3'0) else counter(x + 3'1)

```

```

do
  (** spec: (AG (c <> 1'1))
      & (AG (AF pc = Entry_counter))
      & (AG (AF x-r4 = 3'0) & (AF x-r4 = 3'1)
          & (AF x-r4 = 3'2) & (AF x-r4 = 3'3)
          & (AF x-r4 = 3'4) & (AF x-r4 = 3'5)
          & (AF x-r4 = 3'6) & (AF x-r4 = 3'7)
      )
  **)
  main(x:3):3 = counter(x)

```

The complicated CTL formula specifies several desired properties of the system. First, as before we require that the system not overflow when executing a primitive operation. Second, we require that there are infinitely many entry points to the `counter` function. That is, the program never halts. Reading off this part of the specification in English, “It is globally true on all computation paths (AG) that on all computation paths there is some future state (AF) where `pc = Entry_counter`”. The third part of the conjunction establishes an invariant property (AG) that every possible 3-bit value will be stored internally in `counter`’s formal parameter `x` infinitely often.

In order to determine the register used for `x` in `counter`, we examined the SAFLASM code implementing it:

```

Entry_counter:  MOV    r5 <- x-r4
line_7:         MOV    r6 <- 3'111
line_8:         EQ.1   r7 <- r5, r6
line_9:         BEQZ   r7 line_14
...

```

By inspection, it is clear that `x-r4` is `x` because there is only one input to `counter` (also, the compiler prefixes register names with their declared names in the SAFL source). Note that the un-optimised compilation of SAFL programs to SAFLASM makes it easy to determine the registers that are used for certain values because the instruction sequence is produced in a deterministic, reliable fashion).

Our model checker has no trouble establishing these properties:

```

...
[generating Kripke structure counter.lang ... states: 878]
[elapsed time: 406ms]
[verifying specification...]
CTL cache hit rate: 0.0434782608696%
((AG (c <> 1'1) & AG (AF (pc = Entry_counter))) & ...: yes
[finished counter.lang in 38357ms]

```

However, while this specification captures a some of the desired behaviour of the program, we can do better using the temporal until operator. The specification above does not ensure that the counter increases by one at each iteration. An implementation that counted by 3's (*mod* 8) would also satisfy the specification. Recall that a CTL specification  $A \phi_1 U \phi_2$  is true of a system where on all paths,  $\phi_1$  holds until some unspecified state when  $\phi_2$  holds. We can use this CTL form to specify that the counter increases by 1 at each iteration. The revised specification is as follows (we don't bother to reverify the specifications involving pc and c):

```

fun counter(x:3):3 =
  if (x = 3'7) then counter(3'0) else counter(x + 3'1)
do
  (** spec:
    AG (AF (
      (x-r4 = 3'0) -> A (x-r4 = 3'0) U (x-r4 = 3'1) &
      (x-r4 = 3'1) -> A (x-r4 = 3'1) U (x-r4 = 3'2) &
      (x-r4 = 3'2) -> A (x-r4 = 3'2) U (x-r4 = 3'3) &
      (x-r4 = 3'3) -> A (x-r4 = 3'3) U (x-r4 = 3'4) &
      (x-r4 = 3'4) -> A (x-r4 = 3'4) U (x-r4 = 3'5) &
      (x-r4 = 3'5) -> A (x-r4 = 3'5) U (x-r4 = 3'6) &
      (x-r4 = 3'6) -> A (x-r4 = 3'6) U (x-r4 = 3'7) &
      (x-r4 = 3'7) -> A (x-r4 = 3'7) U (x-r4 = 3'0)
    ))
  **)
main(x:3):3 = counter(x)

```

The specification ensures that whenever the register has a 3-bit value, it holds that value until some state where it changes to a value that is precisely one greater than the previous value, *mod* 8. Again, the model checker successfully verifies the property for this system:

```

[generating Kripke structure counter2.lang ... states: 878]
[elapsed time: 443ms]
[verifying specification...]
CTL cache hit rate: 0.213333333333%
AG (AF (((((((((x-r4 = 3'000 -> (A x-r4 = 3'000 U x... :yes
[finished counter2.lang in 38011ms]

```

Thus, the counter implementation has the desired properties.

### 4.3 Extensions

In addition to the basic system described above, we implemented two extensions. The state explosion problem limits explicit state model checking

to fairly small examples. We have implemented a translation to the input language for the SMV model checker. Additionally, we have extended our model checker to produce counter examples when a specification is proved false.

## 4.4 Verification by translation to SMV

Similar programs can be verified using the translation to SMV. An example this translation for the `counter` program is given in Appendix D. Also, programs that would be too large to verify using explicit state model checking can be translated to SMV and verified. We have used our translator successfully on examples as large as a complete stack machine processor design.

## 4.5 Counter Examples

Recall our first (failed) attempt to verify that a simple SAFL program for addition does not overflow. With the counter example feature turned on, our model checker can produce a sequence of states as a counter example. That is, it produces a computation path where  $\neg (c < 1'1)$ . Here we demonstrate counter example generation in the model checker:

```
[verifying specification...]
CTL cache hit rate: 0.0%
AG (c < 1'1): no

[producing counterexample...]
s0      : [c = 1'0, pc = line_1]
s481    : [c = 1'0, pc = line_2, r3 = 3'1, x-r1 = 3'1,
          y-r2 = 3'7]
s482    : [c = 1'0, pc = line_3, r3 = 3'1, r4 = 3'7,
          x-r1 = 3'1, y-r2 = 3'7]
s483    : [c = 1'0, pc = line_4, r3 = 3'1, r4 = 3'7,
          x-r1 = 3'1, x-r6 = 3'1, y-r2 = 3'7]
s484    : [c = 1'0, pc = line_5, r3 = 3'1, r4 = 3'7,
          x-r1 = 3'1, x-r6 = 3'1, y-r2 = 3'7, y-r7 = 3'7]
s485    : [L_add = line_6, c = 1'0, pc = Entry_add,
          r3 = 3'1, r4 = 3'7, x-r1 = 3'1, x-r6 = 3'1,
          y-r2 = 3'7, y-r7 = 3'7]
s486    : [L_add = line_6, c = 1'0, pc = line_9, r3 = 3'1,
          r4 = 3'7, r8 = 3'1, x-r1 = 3'1, x-r6 = 3'1,
          y-r2 = 3'7, y-r7 = 3'7]
s487    : [L_add = line_6, c = 1'0, pc = line_10, r3 = 3'1,
```

```

      r4 = 3'7, r8 = 3'1, r9 = 3'7, x-r1 = 3'1,
      x-r6 = 3'1, y-r2 = 3'7, y-r7 = 3'7]
s488    : [L_add = line_6, Res_add = 3'0, c = 1'1,
      pc = line_11, r3 = 3'1, r4 = 3'7, r8 = 3'1,
      r9 = 3'7, x-r1 = 3'1, x-r6 = 3'1, y-r2 = 3'7,
      y-r7 = 3'7]
[finished add1.lang in 693ms]

```

In some model checkers it is often unclear how the states produced in a counter example correspond to the original system. However in our model checker, the states correspond to particular SAFLASM instructions. Thus, it is easy to make the connection between a counter example and a SAFLASM program, to identify the bug. Here we see that when `x-r1` has initial value `3'1` and `y-r2` has initial value `3'7`, the instruction at `line_10`:

```
ADD.3    Res_add <- r8, r9
```

causes overflow. Thus, we can adjust the SAFL program as in our second attempt that pads the inputs before performing addition.



## Chapter 5

# Conclusions

*Given the present moment, there are several possibilities for what the next moment may be like – and for each possible next moment, there are several possibilities for the moment after that. Thus the situation takes the form, not of a linear sequence, but of a tree.*

– Saul Kripke<sup>1</sup>

---

The main result of this project is the implementation of a CTL model checker for hardware descriptions written in SAFL. In particular, the following tasks were completed:

- Design and implement an algorithm to extract Kripke structures from SAFL programs. This task includes:
  - Translate SAFL programs to SAFLASM.
  - Implement interpreter to produce Kripke structures from SAFLASM.
- Implement a CTL model checking algorithm.
- Implement an algorithm to produce counter-examples (when possible) for failed specifications.
- Implement a translator that transforms SAFL programs to SMV descriptions.

We showed how temporal properties of SAFL programs can be verified using our tools, and performed some tests to estimate the limit of the size of systems that can be handled with our system. In many cases where an

---

<sup>1</sup>writing to A. N. Prior on tense logics

full explicitly defined Kripke structure would be too large (because of the state explosion problem), the translator to SMV can be used as a model checking back-end to perform the verification. Alternatively, the model can be simplified so that unimportant features are abstracted away.

## 5.1 Future Work

There is a tree of possible directions for future research on verification for SAFL. First, the verification phase of the could be re-written to use a symbolic model checking algorithm. This would circumvent, in many cases, the limitation imposed by the state explosion problem. However, the current implementation's translation to SMV can be used to avoid this limitation when needed. Thus, no new functionality would be gained from this addition. Further, the SMV program is a mature, industrial strength model checker. It is unlikely that a new implementation would improve on its efficiency. Still, it would be beneficial, to integrate a symbolic model checker more tightly with our implementation.

A second possibility for future work is in combining model checking with theorem proving. While model checking is a popular automatic verification technique, the size of systems that can be verified compared is limited when compared to the theorem provers (the tradeoff is that theorem provers require human guidance while model checkers are fully automatic). A current research topic is to combine model checking with theorem proving – simple sub-systems are verified using automatic model checkers and theorem provers are used to combine those results and prove a larger correctness result. This work would involve embedding a temporal logic in a suitable logic and proof environment (*e.g.* the HOL system [Gor85]) and designing a logic to support the kind of compositional reasoning described above.

A related third future direction for the project is the design of a custom specification logic for SAFL programs. While CTL is a very expressive logic, and can handle many temporal properties of programs, it is not able to elegantly handle Hoare-style reasoning about the result of program execution. It would be beneficial to develop a hybrid logic to support both of these paradigms simultaneously. For example, we might wish to prove that a multiplier circuit has some Hoare-style properties (*i.e.* the result is the product of the inputs) and some temporal properties (*i.e.* certain internal values are stable during the computation). Unlike traditional modelling languages (such as the SMV input language) which are little more than descriptions of Kripke structures, SAFL programs look more like programs in a high-level language. Thus, it may be useful to apply techniques developed in verifying programs to SAFL. It will be useful to close the gap between Hoare-style and temporal logics in a logic that supports reasoning about values and about temporal properties of systems.

A final future research direction is in evolving the model checker as SAFL matures. Mycroft and Sharp have already proposed extensions to the original SAFL language to support  $\pi$ -calculus-style channel passing and communication primitives. As SAFL incorporates forms to support explicit (rather than compiler inserted) concurrency, it will be interesting to extend our work in extracting transition systems to support the new forms. We hope that this work is rather easy: as the SAFL code is used almost unmodified, it should be easy to plug in new SAFL front ends and update the necessary syntax tree processors with extra cases for the new forms. In particular, explicit parallelism can be handled by adding a program counter for each parallel process. With concurrency in the language, temporal properties of programs become even more important. Thus a model checker that verifies specifications of SAFL extended thus, would be a very useful tool.

## 5.2 Summary

This project successfully implements a model checker for SAFL. Thus, the basic requirements outlined in Section 2.1 are met. Additionally, we've demonstrated the verification of some properties for simple SAFL programs and described how larger programs can be handled by translation to SMV. Importantly, even though SAFL is a high-level language (and thus the connection between source and transition system is somewhat unclear), using information about the translation to SAFLASM, it is possible to write useful specifications. Finally, we've given some exciting ideas for future research.



# Bibliography

- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Compilers*, C-35(8), 1986. [7](#)
- [Gor85] M. J. C. Gordon. Hol: A Machine Ordered Formulation of Higher Ordered Logic. Technical Report 68, Computer Laboratory, University of Cambridge, July 1985. [40](#)
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. [3](#)
- [HR00] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, Cambridge, 2000. [24](#), [53](#)
- [JGP99] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999. [i](#), [2](#), [24](#), [26](#), [61](#)
- [McM93] K. McMillian. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. [7](#), [61](#), [63](#)
- [MS00] A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *Lecture Notes in Computer Science: Proc. 27th ICALP, vol 1853*. Springer-Verlag, 2000. [i](#), [7](#), [13](#), [18](#), [19](#), [62](#)



# Appendix A

## SAFL

---

This appendix contains the grammar for CTL-annotated SAFL. The following conventions are used: *id* generates a string, *const* an integer with specified width (*e.g.*, 3'7 is the three bit binary expression 111), *int* any integer, and *form\_list* a (possibly empty) list of *forms*.

A SAFL program for model checking has a syntax tree generated by the *prog* production.

```
prog      ::=  fdef_list do spec main( varlist ) : int exp

spec      ::=
            |  (** spec: ctl **)

ctl       ::=  True
            |  False
            |  atom
            |  EX ctl
            |  AX ctl
            |  EF ctl
            |  AF ctl
            |  EG ctl
            |  AG ctl
            |  E ctl U ctl
            |  A ctl U ctl
            |  ctl & ctl
            |  ctl || ctl
            |  ! ctl
            |  ctl -> ctl
            |  ctl <-> ctl
```

<i>atom</i>	::=	<i>id</i> = <i>atomval</i>   <i>id</i> <> <i>atomval</i>   <i>id</i> < <i>const</i>   <i>id</i> > <i>const</i>   <i>id</i> <= <i>const</i>   <i>id</i> >= <i>const</i>
<i>atomval</i>	::=	<i>id</i>   <i>const</i>   ?
<i>fdef</i>	::=	<b>fun</b> <i>id</i> ( <i>varlist</i> ) : <i>int</i> = <i>exp</i>
<i>varlist</i>	::=	<i>id</i> : <i>int</i> <i>varlist</i>
<i>exp</i>	::=	<b>if</b> <i>exp</i> <b>then</b> <i>exp</i> <b>else</b> <i>exp</i>   <b>case</b> <i>exp</i> <b>of</b> <i>match_list</i> <b>default</b> => <i>exp</i>   <i>iexp</i>   <i>aexp</i>
<i>iexp</i>	::=	<i>iexp</i> <i>binary_op</i> <i>iexp</i>   <i>unary_op</i> <i>iexp</i>   <i>iexp</i> [ <i>int</i> : <i>int</i> ]   <i>iexp</i> [ <i>int</i> ]   <i>aexp</i>
<i>binary_op</i>	::=	=   <>   <   >   <=   >=   +   *   /   >>   >>          &&   &
<i>unary_op</i>	::=	~   ~~
<i>aexp</i>	::=	<i>id</i>   <i>const</i>   ( <i>sexp</i> )   <i>id</i> ( <i>exp_list</i> )   <b>join</b> ( <i>exp_list</i> )   <b>let</b> <i>dec_list</i> <b>in</b> <i>sexp</i> <b>end</b>
<i>sexp</i>	::=	<i>exp</i>   <i>exp</i> ; <i>exp</i>



```
dec      ::= id : int = iexp  
  
match    ::= exp => exp_nc  
  
exp_nc    ::= iexp  
            | if exp_nc then exp_nc else exp_nc
```



## Appendix B

# SAFLasm

---

Here we give the full syntax and semantics for SAFLASM programs. We also give the complete translation of SAFL into SAFLASM.

### B.1 Syntax

```
code      ::=  
            | inst ; code  
inst      ::= label : MOV op <-op  
            | label : JMP label  
            | label : JSR op label  
            | label : RET op  
            | label : BEQZ op label  
            | label : binary_op op <-op op  
            | label : unary_op op <-op  
  
binary_op ::= ADD | SUB | MULT | DIV  
            | RSHIFT | LSHIFT | AND | OR  
            | XOR | EQ | NEQ | LT  
            | GT | LEQ | GEQ | LOR  
            | LAND | LXOR  
  
unary_op  ::= NOT | LNOT  
  
op        ::= id  
            | const  
  
label     ::= id
```

## B.2 Operational Semantics

As the instructions in SAFLASM are simple, the operational semantics for the language is correspondingly straight-forward. Here we give a semantics for processing SAFLASM *code*. The rules assume a static list of instruction code,  $C$ , and make use of an environment  $E$  which maps registers to values:  $?$ , labels, and integers. The notation  $E[x \leftarrow v]$  specifies an environment that maps  $x$  to  $v$  and otherwise behaves like  $E$ .  $E(x)$  looks up the value of  $x$  stored in the environment if  $x$  is the name of a register. If  $x$  is a constant, then  $E(x)$  simply gives the value of the constant. Similarly,  $E(?)$  is  $?$ .

First we define an auxiliary predicate for looking up the instructions starting with a label  $l$  (we use but do not define  $label(I)$  which gives the label of instruction  $I$ ).

$$\frac{label(I) = l}{nextInstr(l, I) = I}$$

$$\frac{label(I) \neq l}{nextInstr(l, I; I') = nextInstr(l, I')}$$

Thus, we can lookup the list of instructions starting with an instruction labelled  $lbl$  like this:

$$nextInstr(lbl, C)$$

The operational semantics is defined for each form in SAFLASM:

$$\begin{aligned} \langle 1 : \text{MOV } op_1 \leftarrow op_2 ; I', E \rangle &\Downarrow \langle I', E[op_1 \leftarrow E(op_2)] \rangle \\ \langle 1 : \text{JMP } label_1 ; I', E \rangle &\Downarrow \langle nextInstr(label_1, C), E \rangle \\ \langle 1 : \text{JSR } op_1 \text{ } label_2 ; I', E \rangle &\Downarrow \langle nextInstr(label_2, C), \\ &\quad E[op_1 \leftarrow nextInstr(l, C)] \rangle \\ \langle 1 : \text{RET } op_1 ; I', E \rangle &\Downarrow \langle nextInstr(E(op_1), C), E \rangle \\ \langle 1 : \text{BEQZ } op_1 \text{ } label_1 ; I', E \rangle &\Downarrow \langle I', E \rangle \text{ if } E(op_1) \neq 0 \\ \langle 1 : \text{BEQZ } op_1 \text{ } label_1 ; I', E \rangle &\Downarrow \langle nextInstr(label_1, C), E \rangle \text{ if } E(op_1) = 0 \\ \langle 1 : \text{PRIMOP } op_1 \leftarrow op_2 \text{ } op_3 ; I', E \rangle &\Downarrow \langle I', E[op_1 \leftarrow op_2 \text{ } primop \text{ } op_3] \rangle \\ \langle 1 : \text{PRIMOP } op_1 \leftarrow op_2 ; I', E \rangle &\Downarrow \langle I', E[op_1 \leftarrow primop \text{ } op_2] \rangle \end{aligned}$$

### B.3 Translation of SAFL to SAFLASM

$$\begin{aligned}
\llbracket c \rrbracket l &= \text{MOV } l \leftarrow c \\
\llbracket x \rrbracket l &= \text{MOV } l \leftarrow M_x \\
\llbracket \text{if}(\langle l_1 \rangle : e_1) \text{ then } e_2 \text{ else } e_3 \rrbracket l &= \begin{array}{l} \llbracket e_1 \rrbracket l_1 \\ \text{BEQZ } l_1 \text{ lfalse} \\ \llbracket e_2 \rrbracket l \\ \text{JMP } l_{\text{next}} \\ \text{lfalse: } \llbracket e_3 \rrbracket l \\ \text{lnext: } \dots \end{array} \\
\llbracket f(\langle l_1 \rangle e_1, \dots \langle l_n \rangle e_n) \rrbracket l &= \begin{array}{l} \llbracket e_1 \rrbracket l_1 \\ \dots \\ \llbracket e_n \rrbracket l_n \\ \text{MOV } M_{f\text{formals}_1} \leftarrow l_1 \\ \dots \\ \text{MOV } M_{f\text{formals}_n} \leftarrow l_n \\ \text{JSR } L_f \text{ Entry}_f \\ \text{MOV } l \leftarrow \text{Result}_f \end{array} \\
\llbracket f(\langle l_1 \rangle e_1, \dots \langle l_n \rangle e_n) \rrbracket l &= \begin{array}{l} \llbracket e_1 \rrbracket l_1 \\ \dots \\ \llbracket e_n \rrbracket l_n \\ \text{MOV } M_{f\text{formals}_1} \leftarrow l_1 \\ \dots \\ \text{MOV } M_{f\text{formals}_n} \leftarrow l_n \\ \text{JMP } \text{Entry}_f \end{array} \\
\llbracket a(\langle l_1 \rangle e_1, \dots \langle l_n \rangle e_n) \rrbracket l &= \begin{array}{l} \llbracket e_1 \rrbracket l_1 \\ \dots \\ \llbracket e_n \rrbracket l_n \\ \text{PRIMOP}_a \text{ } l \leftarrow e_1 \dots e_n \end{array} \\
\llbracket \langle l_1 \rangle e_1; e_2 \rrbracket l &= \begin{array}{l} \llbracket e_1 \rrbracket l_1 \\ \llbracket e_2 \rrbracket l \end{array} \\
\llbracket \text{let var } \langle l_1 \rangle x_1 : w_1 = e_1 \dots \\ \text{var } \langle l_n \rangle x_n : w_n = e_n \text{ in } e \text{ end} \rrbracket l &= \begin{array}{l} \llbracket e_1 \rrbracket l_1 \\ \dots \\ \llbracket e_n \rrbracket l_n \\ \llbracket e \rrbracket l \end{array} \\
\llbracket \text{fun } f(x_1 : w_1, \dots x_n : w_n) : w = e \rrbracket &= \text{Entry}_f : \begin{array}{l} \llbracket e \rrbracket \text{Res}_f \\ \text{RET } L_f \end{array}
\end{aligned}$$



## Appendix C

# Model Checking Algorithm

---

This appendix contains the predicate that is the basis for our implementation of the verification phase in our project. It determines the set of states in a model satisfying a CTL formula by either simple set operations, rewriting to another CTL formula, or a least fixed point computation. It is linear in the size of the proposition and quadratic in the size of the model and is adopted from [HR00].

Assume that the model is an explicitly defined Kripke structure  $M = (S, S_0, R, L)$  and that  $\phi$  is a CTL formula. The *sat* predicate returns the following set:  $\{s \in S \mid M, s \models \phi\}$ . It is defined inductively on the structure of  $\phi$ .

$$\begin{aligned} \text{sat}(\top) &= S \\ \text{sat}(\perp) &= \emptyset \\ \text{sat}(p) &= \{s \mid s \in S \ \& \ s \in L(p)\} \\ \text{sat}(\neg\phi) &= S - \text{sat}(\phi) \\ \text{sat}(\phi_1 \vee \phi_2) &= \text{sat}(\phi_1) \cup \text{sat}(\phi_2) \\ \text{sat}(\phi_1 \ \& \ \phi_2) &= \text{sat}(\phi_1) \cap \text{sat}(\phi_2) \\ \text{sat}(\phi_1 \Rightarrow \phi_2) &= \text{sat}(\neg\phi_1 \vee \phi_2) \\ \text{sat}(\phi_1 \equiv \phi_2) &= \text{sat}((\neg\phi_1 \vee \phi_2) \ \& \ (\neg\phi_2 \vee \phi_1)) \\ \text{sat}(\text{EX } \phi) &= \{s \mid s \in S \ \& \ \exists s'. R(s, s') \ \& \ s' \in \phi\} \\ \text{sat}(\text{AX } \phi) &= \text{sat}(\neg\text{EX } \neg\phi) \\ \text{sat}(\text{EF } \phi) &= \text{sat}(\text{E } \top \text{ U } \phi) \\ \text{sat}(\text{AF } \phi) &= \mu Z. \phi \vee \text{AX } Z \\ \text{sat}(\text{EG } \phi) &= \text{sat}(\neg\text{AF } \neg\phi) \\ \text{sat}(\text{AG } \phi) &= \text{sat}(\neg\text{EF } \neg\phi) \\ \text{sat}(\text{E } \phi_1 \text{ U } \phi_2) &= \mu Z. \phi_2 \vee (\phi_1 \ \& \ \text{EX } Z) \\ \text{sat}(\text{A } \phi_1 \text{ U } \phi_2) &= \text{sat}(\neg(\text{E}[\neg\phi_1 \text{ U } (\neg\phi_1 \ \& \ \neg\phi_2)] \vee \text{EG } \neg\phi_2)) \end{aligned}$$





## Appendix D

# Example: SMV Translation

---

Here we give a sample translation of a simple SAFL program to SMV. We keep the example short, while illustrating the general technique. This simple program:

```
fun counter(x:3):3 =
  if (x = 3'7) then counter(3'0) else counter(x + 3'1)
do
(** spec: AG (AF (
  (x-r4 = 3'0) -> A (x-r4 = 3'0) U (x-r4 = 3'1) &
  (x-r4 = 3'1) -> A (x-r4 = 3'1) U (x-r4 = 3'2) &
  (x-r4 = 3'2) -> A (x-r4 = 3'2) U (x-r4 = 3'3) &
  (x-r4 = 3'3) -> A (x-r4 = 3'3) U (x-r4 = 3'4) &
  (x-r4 = 3'4) -> A (x-r4 = 3'4) U (x-r4 = 3'5) &
  (x-r4 = 3'5) -> A (x-r4 = 3'5) U (x-r4 = 3'6) &
  (x-r4 = 3'6) -> A (x-r4 = 3'6) U (x-r4 = 3'7) &
  (x-r4 = 3'7) -> A (x-r4 = 3'7) U (x-r4 = 3'0)
)) **))
main(x:3):3 = counter(x)
```

is compiled SAFLASM instructions:

```
line_1:      MOV      r2 <- x-r1
line_2:      MOV      x-r4 <- r2
line_3:      JSR      L_counter, Entry_counter
line_4:      MOV      r3 <- Res_counter
line_5:      JMP      line_5

Entry_counter: MOV      r5 <- x-r4
```

```

line_7:      MOV      r6 <- 3'111
line_8:      EQ.1     r7 <- r5, r6
line_9:      BEQZ     r7 line_14
line_10:     MOV      r8 <- 3'000
line_11:     MOV      x-r4 <- r8
line_12:     JMP      Entry_counter
line_13:     JMP      line_19
line_14:     MOV      r10 <- x-r4
line_15:     MOV      r11 <- 3'001
line_16:     ADD.3    r12 <- r10, r11
line_17:     MOV      x-r4 <- r12
line_18:     JMP      Entry_counter
line_19:     RET      L_counter

```

The SMV description is easily generated from this syntax.

```
MODULE main
```

```
VAR
```

```

pc : {line_1, line_2, line_3, line_4, line_5,
      Entry_counter, line_7, line_8, line_9, line_10,
      line_11, line_12, line_13, line_14, line_15,
      line_16, line_17, line_18, line_19};
L_counter : {line_1, line_2, line_3, line_4, line_5,
             Entry_counter, line_7, line_8, line_9, line_10,
             line_11, line_12, line_13, line_14, line_15,
             line_16, line_17, line_18, line_19};
Res_counter : array 2..0 of boolean;
r3 : array 2..0 of boolean;
r2 : array 2..0 of boolean;
r5 : array 2..0 of boolean;
r6 : array 2..0 of boolean;
r7 : array 0..0 of boolean;
r8 : array 2..0 of boolean;
r10 : array 2..0 of boolean;
r11 : array 2..0 of boolean;
r12 : array 2..0 of boolean;
x-r1 : array 2..0 of boolean;
x-r4 : array 2..0 of boolean;

```

```
ASSIGN
```

```

init(pc) := line_1;

next(pc) :=
  case

```

```

(pc = line_1) : line_2;
(pc = line_2) : line_3;
(pc = line_3) : Entry_counter;
(pc = line_4) : line_5;
(pc = line_5) : line_5;
(pc = Entry_counter) : line_7;
(pc = line_7) : line_8;
(pc = line_8) : line_9;
(pc = line_9) :
    case
        (r7 = [0]) : line_14;
        (!r7 = [0]) : line_10;
    esac;
(pc = line_10) : line_11;
(pc = line_11) : line_12;
(pc = line_12) : Entry_counter;
(pc = line_13) : line_19;
(pc = line_14) : line_15;
(pc = line_15) : line_16;
(pc = line_16) : line_17;
(pc = line_17) : line_18;
(pc = line_18) : Entry_counter;
(pc = line_19) : L_counter;
(1) : pc;
esac;

next(r3) :=
    case
        (pc = line_4) : Res_counter;
        (1) : r3;
    esac;

next(L_counter) :=
    case
        (pc = line_3) : line_4;
        (1) : L_counter;
    esac;

next(Res_counter) :=
    case
        (1) : Res_counter;
    esac;

next(r10) :=

```

```

    case
      (pc = line_14) : x-r4;
      (1) : r10;
    esac;

next(r11) :=
  case
    (pc = line_15) : [0, 0, 1];
    (1) : r11;
  esac;

next(r12) :=
  case
    (pc = line_16) : r10 + r11;
    (1) : r12;
  esac;

next(r2) :=
  case
    (pc = line_1) : x-r1;
    (1) : r2;
  esac;

next(r5) :=
  case
    (pc = Entry_counter) : x-r4;
    (1) : r5;
  esac;

next(r6) :=
  case
    (pc = line_7) : [1, 1, 1];
    (1) : r6;
  esac;

next(r7) :=
  case
    (pc = line_8) : r5 = r6;
    (1) : r7;
  esac;

next(r8) :=
  case
    (pc = line_10) : [0, 0, 0];

```

```

        (1) : r8;
    esac;

next(x-r1) :=
    case
        (1) : x-r1;
    esac;

next(x-r4) :=
    case
        (pc = line_2) : r2;
        (pc = line_11) : r8;
        (pc = line_17) : r12;
        (1) : x-r4;
    esac;

SPEC
AG (AF (((((((
    (x-r4 = [0,0,0] -> (E x-r4 = [0,0,0] U x-r4 = [0,0,1]))
    & (x-r4 = [0,0,1] -> (E x-r4 = [0,0,1] U x-r4 = [0,1,0]))
    & (x-r4 = [0,1,0] -> (E x-r4 = [0,1,0] U x-r4 = [0,1,1]))
    & (x-r4 = [0,1,1] -> (E x-r4 = [0,1,1] U x-r4 = [1,0,0]))
    & (x-r4 = [1,0,0] -> (E x-r4 = [1,0,0] U x-r4 = [1,0,1]))
    & (x-r4 = [1,0,1] -> (E x-r4 = [1,0,1] U x-r4 = [1,1,0]))
    & (x-r4 = [1,1,0] -> (E x-r4 = [1,1,0] U x-r4 = [1,1,1]))
    & (x-r4 = [1,1,1] -> (E x-r4 = [1,1,1] U x-r4 = [0,0,0]))
)))
))

```



## Appendix E

# Project Proposal

---

**Project Supervisors:** Dr M. Gordon [and Dr A. Mycroft]

**Director of Studies:** Dr N. Dodgson

**Project Overseer:** Dr M. Gordon

### Introduction

Formal verification has become an important area of research as embedded, real-time, and information systems in general, increasingly play important roles in everyday life. There are many situations where verifying that a system correctly conforms to a formal specification is a desirable practical and theoretical result. Unfortunately traditional formal methods are often unwieldy. For modern systems such as large integrated circuits, the complexity of proving formal properties can be prohibitive for even a clever software or hardware engineer. However, work in verification has made it possible to prove formal properties of a growing number of systems automatically. The goal of this project is to explore one such technique, model checking, and implement a model checker by integrating results from the FLaSH project at the Computer Laboratory.

*Model checking* [McM93, JGP99] is a technique for proving properties about concurrent finite state systems automatically. A model checker operates in the following way. First, the device, program, or protocol being checked is modelled as a finite automaton (typically a particular kind of automaton known as a Kripke structure). Then the property to be proved is expressed in a *temporal logic* – a family of logics that can express relationships involving time, though without an explicit representation of time.

Model checking then, simply involves determining for which states in the automaton the formal property holds. If the initial configurations of the system are in this set, then the property is proved for the system. The major advantage of model checking when compared with other reasoning techniques such as mechanised theorem proving is that all of the actual checking can be done completely automatically. This technique has been very successful at verifying a large class of systems, typically hardware circuits. However, while the specific algorithms and representations used in model checking have improved the complexity of circuits have grown even faster. A problem known as *state explosion* limits the state-of-the-art in model checking – for complex parallel systems, the number of states in the model checking structures become prohibitively large.

SAFL [MS00] is a concurrent, statically allocated, functional language developed by Sharp and Mycroft as part of the FLaSH project at the Computer Laboratory and AT&T Labs. The primary goal of the project is to explore using a high-level functional language to describe hardware systems. Accordingly, the SAFL optimising compiler emits a description of a design description in Verilog, not a binary executable. This close connection between high-level SAFL programs and hardware makes it ideal as a modelling language for model checking.

While model checking itself is not new, using a high-level functional language to describe the finite state system presents some interesting possibilities for extensions of the basic technique.

## Proposal

The broad goal of this project is to research hardware verification using SAFL. The completion of a basic-level project entails implementing a model checker with SAFL as the modelling language. I will implement the project using a language from the ML family on UNIX. The following steps describe the work needed to complete this goal:

1. Parse SAFL programs (might use an existing parser).
2. Implement an algorithm to build a Kripke structure from a SAFL parse tree. This involves coming up with a way to identify the starting states and the relation that allows the system to transition between states from the SAFL program. From these, a Kripke structure can be easily constructed.
3. Implement a simple model checker for verifying properties of SAFL programs encoded in the temporal logic CTL. In this step I will have to write code to check each syntactic form in CTL.



4. Perform some empirical tests of properties in CTL against SAFL programs. As model checking has historically been quite limited in the complexity of systems that can be checked (because of state explosion), it will be interesting to evaluate how my implementation compares to previous work.

## Extensions

I have defined a fairly basic project above. However, I hope that I will have time to extend the project with some extra features. Some of these follow directly from the project and should be fairly straight forward. Others take the project in a different, interesting direction. Here I present a list of possible extensions.

**Symbolic model checker:** McMillan’s thesis [McM93] describes a technique for model checking that can be used to verify much more complex systems than was traditionally feasible. The key contribution of his thesis was the observation that the explicit representation of circuits using Kripke structures is a major bottleneck for model checking. As an alternative, he proposes a symbolic representation based on Binary Decision Diagrams. Because a symbolic model checker does not have to simultaneously keep a representation of each state, the effects of the state explosion problem can be reduced. Implementing a symbolic model checker would be a major improvement over the naive version, at the cost of re-writing the core of my implementation.

**Semantics for SAFL transformations:** As part of the existing SAFL implementation, the compiler performs several optimisations that transform SAFL programs. Currently these transformations have not been incorporated into any formal semantics. An interesting theoretical extension is to define such a semantics, and verify the correctness of the transformation stage itself by proving that properties checked on source SAFL programs are preserved by the transformations. A similar extension would be to compare the results proving properties on SAFL source programs using my model checker with those proved by an industrial-grade model checker (such as SMV, the result of McMillan’s thesis work) on the Verilog description produced by the SAFL compiler.

**Integrate theorem proving with model checking:** One branch of recent research uses model checking in conjunction with theorem proving to extend the range of circuits that can be reasoned about. The idea is to check the the relatively simple components of a complex system using model checking and the entire ensemble using more powerful techniques such as theorem proving. Using this technique, properties

of the entire system can be verified. Unlike pure model checking, the entire verification process may not be completely automatic. Still, enough of the work is automated so it is hoped that more systems could be verified in practice using this technique. This extension is an active area of research in several different laboratories around the world. Thus, an integrated model checker/theorem prover is probably beyond the scope of a CST project. However, some interesting work could be done to build a prototype of how such a system, and comment on the challenges involved.

## Starting Point

Having never done any work in formal verification before, I have a large amount of reading to do to get familiar with the field. My only relevant background is a general interest in logic, theory, and semantics including several undergraduate courses taken at Williams College and Trinity College Dublin as well as an undergraduate dissertation completed at Williams. However, as model checking is a fairly self-contained, well known technique, I hope to keep the amount of background needed to get started, to a manageable level.

## Work Plan

Project work commences on 19th Oct 2001.

### Michaelmas Term

The first two weeks of the project will be spent continuing to read about model checking and SAFL as well as getting familiar with the details of existing implementations of those systems. By the end of this exploratory period I will identify exactly which algorithms and structures I wish to use to implement my project. I will also know which tools (parsers, data structures, etc) have already been built by the FLaSH groups and might be useful.

In the second two week period I will spend some time carefully designing the algorithms and data structures as they would work starting from SAFL programs represented in ML.

Towards the end of Michaelmas term, the beginnings of the project implementation will start to take shape. At the very least, a front-end including parsing and the solid steps towards extraction of Kripke structures from SAFL programs should be completed by the end of term.

## **Lent Term**

Some time over the Christmas vacation and the first two weeks of Lent term will be spent completing and testing a basic implementation of a model checker including coding the steps to actually perform the checks for CTL predicates. Depending on my progress by this point, this could either be a naive implementation, or a more sophisticated one (perhaps beginning the implementation of a symbolic model checker).

By the project report deadline on February 1, 2002 I will be able to present results from model checking simple SAFL programs. The next four weeks will be spent testing, gathering any empirical results, and extending the model checker as described previously. Towards the end of this testing/extension period I will start writing the dissertation. The last two weeks of Lent will be entirely devoted to writing, with the goal of having something near a final draft by the start of Easter vacation.

## **Easter Term**

Easter term is reserved for responding to comments from readers and possibly writing up any further extensions or empirical results obtained over the Easter vacation. This leaves plenty of time for editing, proof reading, and practical matters such as printing and binding before the dissertation deadline on 17th May 2002.